INFORMATION INFRASTRUCTURES IN DISTRIBUTED ENVIRONMENTS: ALGORITHMS FOR MOBILE NETWORKS AND RESOURCE ALLOCATION

A Dissertation

by

HYUN CHUL CHUNG

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Jennifer L. Welch
Committee Members,	Riccardo Bettati
	Anxiao Jiang
	Alexander Sprintson
Head of Department,	Nancy Amato

December 2013

Major Subject: Computer Science

ABSTRACT

A distributed system is a collection of computing entities that communicate with each other to solve some problem. Distributed systems impact almost every aspect of daily life (e.g., cellular networks and the Internet); however, it is hard to develop services on top of distributed systems due to the unreliable nature of computing entities and communication. As handheld devices with wireless communication capabilities become increasingly popular, the task of providing services becomes even more challenging since dynamics, such as mobility, may cause the network topology to change frequently. One way to ease this task is to develop collections of *information infrastructures* which can serve as building blocks to design more complicated services and can be analyzed independently.

The first part of the dissertation considers the dining philosophers problem (a generalization of the mutual exclusion problem) in static networks. A solution to the dining philosophers problem can be utilized when there is a need to prevent multiple nodes from accessing some shared resource simultaneously. We present two algorithms that solve the dining philosophers problem. The first algorithm considers an asynchronous message-passing model while the second one considers an asynchronous shared-memory model. Both algorithms are crash fault-tolerant in the sense that a node crash only affects its local neighborhood in the network. We utilize failure detectors (system services that provide some information about crash failures in the system) to achieve such crash fault-tolerance. In addition to crash fault-tolerance, the first algorithm provides fairness in accessing shared resources and the second algorithm tolerates transient failures (unexpected corruptions to the system state). Considering the message-passing model, we also provide a reduction such that given

a crash fault-tolerant solution to our dining philosophers problem, we implement the failure detector that we have utilized to solve our dining philosophers problem. This reduction serves as the first step towards identifying the minimum information regarding crash failures that is required to solve the dining philosophers problem at hand.

In the second part of this dissertation, we present information infrastructures for mobile ad hoc networks. In particular, we present solutions to the following problems in mobile ad hoc environments: (1) maintaining neighbor knowledge, (2) neighbor detection, and (3) leader election. The solutions to (1) and (3) consider a system with perfectly synchronized clocks while the solution to (2) considers a system with bounded clock drift. Services such as neighbor detection and maintaining neighbor knowledge can serve as a building block for applications that require point-to-point communication. A solution to the leader election problem can be used whenever there is a need for a unique coordinator in the system to perform a special task.

DEDICATION

To my wife, my son, and my parents.

ACKNOWLEDGEMENTS

Above all, I would like to express my deepest appreciation to my advisor Dr. Jennifer L. Welch for her guidance and support during my graduate studies. Whenever I reached a research roadblock, she has guided me towards the right path and encouraged me to move forward to reach my goal. I have the greatest respect for her personality and work ethics.

I would also like to thank my committee members Drs. Riccardo Bettati, Andrew Jiang, and Alex Sprintson for their support in improving the quality of my research.

During my studies at Texas A&M, I had the privilege to work with wonderful research group members in the Distributed Computing group, Parasol Laboratory: Dr. Hyunyoung Lee, Dr. Srikanth Sastry, Dr. Saira Viqar, Gautam Roy, Erica Wang, Edward Talmage, Mira Radeva, Whitney Maguffee, Dr. Josef Widder, and Dr. Peter Robinson. They were generous enough to have lengthy discussions with me even when I raised a tedious topic. They are more than my colleagues; they are my friends.

Last but not least, I would like to thank my parents, my wife Ji Yun, and my son Won Il for their support, patience and love.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

1. INTRODUCTION

A distributed system is a collection of computing entities that communicate with each other to solve some problem. Distributed systems impact almost every aspect of daily life (e.g., cellular networks and the Internet); however, it is hard to develop services on top of distributed systems due to the unreliable nature of computing entities and communication; node and link failures might occur due to hardware deterioration and message transmissions might be delayed due to message collisions and interference. As handheld devices with wireless communication capabilities become increasingly popular, the task of providing services becomes even more challenging since dynamics, such as mobility, may cause the network topology to change frequently. One way to ease this task is to develop collections of *information infrastructures* which can serve as building blocks to design more complicated services and can be analyzed independently.

This dissertation is divided into two parts based on the dynamicity of the system. The first part presents information infrastructures for static networks and second part presents information infrastructures for mobile ad hoc networks — networks in which nodes are mobile and no fixed physical infrastructures, such as base stations, exist.

In the remainder of this section, we discuss the motivation, research goal, and the novelty of the information infrastructures considered in this dissertation. We also provide the organization of this dissertation.

1.1 Motivation and Research Goal

Our goal is to provide useful information infrastructures that serve as building blocks for other problems to utilize. In particular, we consider the the following information infrastructures in this dissertation:

- Dining philosophers (generalized mutual exclusion),
- Maintaining neighbor knowledge,
- Neighbor detection, and
- Leader election.

We consider the dining philosophers problem in a network where nodes are static. Maintaining neighbor knowledge, neighbor detection, and leader election are considered in a mobile ad hoc environment.

An algorithm that solves the dining philosophers problem can be utilized when there is a need to prevent multiple nodes from accessing some shared resource simultaneously. Services such as neighbor detection and maintaining neighbor knowledge can serve as building blocks for applications that require point-to-point communication. A solution to the leader election problem can be used whenever there is a need for a unique coordinator in the system to perform a special task.

We examine each information infrastructure in more detail in the remainder of this section.

1.1.1 Fault-Tolerant Dining Philosophers

The dining philosophers problem [24, 49], or simply dining, is a distributed resource allocation problem, in which each node repeatedly needs simultaneous exclusive access to a set of shared resources in order to enter a special part of its code, called the critical section. The sharing pattern is described by an arbitrary "conflict" graph, each edge of which represents a resource shared by the two nodes corresponding to the endpoints of the edge.

In large scale and long-lived systems, the likelihood of some node failing at some point is high, thus sparking interest in fault-tolerant versions of the dining problem. The ideal case would be for the dining algorithm to isolate each crashed node such

that it does not impact any other correct nodes in the system. If the ideal case is impossible to achieve, restricting the impact of the crash failure to a local neighborhood would still be desirable. *Failure locality* [13, 14] is a metric that realizes this concept; it is the maximum distance in the conflict graph between a crashed node p and any other node that is blocked from entering its critical section.

In this dissertation, we provide two problem definitions that extend the classical dining philosophers problem in asynchronous systems — systems in which no bounds on message delivery time and relative process speed exist. The first problem definition considers failure locality 1 and fairness in accessing shared resources in an asynchronous message passing model. The second definition also considers failure locality 1, however, instead of fairness, it considers the presence of transient failures (unexpected corruptions to the system state) in an asynchronous shared memory model. In the asynchronous message passing model, nodes communicate by sending and receiving messages through communication links. However, in the asynchronous shared memory model, nodes communicate via performing read/write operations on shared memory objects.

We also present two novel distributed algorithms, each algorithm solving each one of the above mentioned problems. Each of the two algorithms is the first algorithm for its corresponding problem.

Choy and Singh [14] showed that any dining algorithm implemented on an asynchronous system must have failure locality at least 2. To circumvent this lowerbound, our algorithms utilize failure detectors — system services that provide some information about crash failures in the system — to achieve failure locality 1. The failure detectors that we use are at most as powerful as the ones that are used to solve failure-locality-0 dining [68, 69, 62, 65]; we are trading off failure locality to use failure detectors that seemingly provide less information regarding crash failures in the

system.

Considering the message passing model, we also provide a reduction such that given a crash fault-tolerant solution to our dining philosophers problem, we implement the failure detector that we have utilized to solve our dining philosophers problem. This reduction serves as the first step towards identifying the minimum information regarding crash failures that is required to solve the dining philosophers problem at hand.

1.1.2 Maintaining Neighbor Knowledge and Neighbor Detection

In wireless ad hoc networks, a fundamental problem for a node is to keep track of its set of nearby nodes (neighbors). In this case, the challenge is to deal with wireless broadcast interference and collisions. Difficulties are added when nodes are mobile since frequent change of network topology requires the process of identifying nearby neighbors to be ongoing. This problem of identifying nearby nodes serves as a basis of distributed primitives such as routing ([60, 64]), location services ([2]), and distributed token circulation ([50]).

In this dissertation, we present two solutions to the problem of identifying nearby nodes. Each of the two solutions is the first solution for its corresponding system model.

1.1.2.1 Maintaining Neighbor Knowledge in Road Networks

Assuming that, initially, each node knows its neighbors, we consider the problem of maintaining neighbor knowledge where node mobility is restricted to a two-dimensional road network. A road network is a collection of one-dimensional lines that may intersect each other. We can view vehicular ad hoc networks as nodes (vehicles) with wireless communication capabilities moving on a road network.

For nodes to exchange neighbor information, we construct a deterministic collision-

free broadcast schedule which utilizes time division multiplexing and geographical segmentation. Under certain constraints, our broadcast schedule tolerates node movement on the road network while providing deterministic guarantees for each node to maintain its dynamically changing set of neighbors.

We also provide a bound on how fast messages can travel in the road network given our broadcast schedule. If nodes broadcast timestamped messages periodically, then this bound is particularly useful when some node p needs to estimate the distance between itself and another node q on the road network: when node p receives a message m from q at time t, it can calculate an upper bound on the distance between itself and q using time t, the timestamp value in m, message broadcast period, and the bound on speed of message propagation.

Considering vehicular networks, we often observe that vehicles move in clusters (due to, for instance, traffic lights). To this matter, we consider grouping nodes on the road network into clusters and show that, under certain conditions, neighbor knowledge is maintained when two different clusters move close to each other.

Our solution is an extension of the one-dimensional case in [27] and [70] to two-dimensional space. One might think that extending the results in [27] and [70] would be trivial since a road network is a collection of straight lines. However, since we consider that the segmentation of each individual line is independent from each other and there is no restriction on where lines intersect, providing analysis with respect to intersection points becomes non-trivial. We perform a rigorous case-by-case analysis whenever there is a need to consider intersection points in our analysis.

The material regarding maintaining neighbor knowledge in road networks has been published in [18].

1.1.2.2 Neighbor Detection in Mobile Ad Hoc Networks

Discovering neighboring nodes in mobile ad hoc networks becomes fully meaningful when nodes identify neighbors that they can actually communicate with. We provide a solution where, for any given node p, nodes that are identified as neighbors of p can perform reliable point-to-point communication with p.

We build our solution on top of the abstract MAC layer [45] which handles the scheduling of wireless transmissions (collision detection and contention management) and provides message delay bounds for our solution to utilize. In this way, we can focus on the algorithmic aspects of our solution without dealing with the scheduling of wireless transmissions.

Our solution is based on nodes generating periodic hello messages where each hello message contains sufficient information to help nodes to identify neighbors with which they can communicate in a point-to-point manner. To make the problem more suitable for a wider variety of situations, we consider clocks with bounded drift rates.

To the best of our knowledge, our solution and the algorithm in [19] are the only neighbor detection algorithms that consider the use of the abstract MAC layer. The neighbor detection algorithm in [19] relies on geographical segmentation and each mobile node to have future knowledge on joining or leaving a geographical segment. However, our solution does not rely on geographical segmentation and does not require mobile nodes to know their future whereabouts.

1.1.3 Leader Election in Mobile Ad Hoc Networks

Leader election is a process of identifying a unique existing node in the system. In this dissertation, we consider the problem of electing a leader within a fixed region where (1) nodes are mobile, (2) nodes communicate via wireless broadcast, and (3)

clocks are synchronized. The advantage of considering a region is that leader election can be performed by nodes that are relatively close to each other. If we consider performing leader election among nodes that are connected but far apart from each other, the amount of time that it takes to elect a leader and knowing which node became the leader can be significant.

We assume that a message broadcast in the region is relayed through at most a fixed number of hops to nodes that stayed in the region for a sufficiently long period of time (in this case, we say that the region has a bounded communication diameter). By having this assumption, any pair of mobile nodes in the region are not required to be connected at all times (network partitions are allowed within the region) which is a better fit for mobile ad hoc networks

We provide a novel problem definition, called the Regional Consecutive Leader Election (RCLE) problem, by extending the classical leader election problem to the ever changing environment of mobile ad hoc networks.

We prove that any algorithm in our model requires $\Omega(Dn)$ synchronous rounds to solve the RCLE problem, where D is the communication diameter of the network and n is the total number of nodes. We then present the first asymptotically optimal fault-tolerant RCLE algorithm that elects a new leader whenever no leader is present in the region and there exists at least one node in the region. Since our leader election algorithm assumes that the region has a bounded communication diameter, the question arises how to ensure that the region has a bounded communication diameter? To answer this question, we provide a novel condition on node mobility that ensures the existence of a bounded communication diameter within the region.

The material regarding regional consecutive leader election has been published in [15, 16, 17].

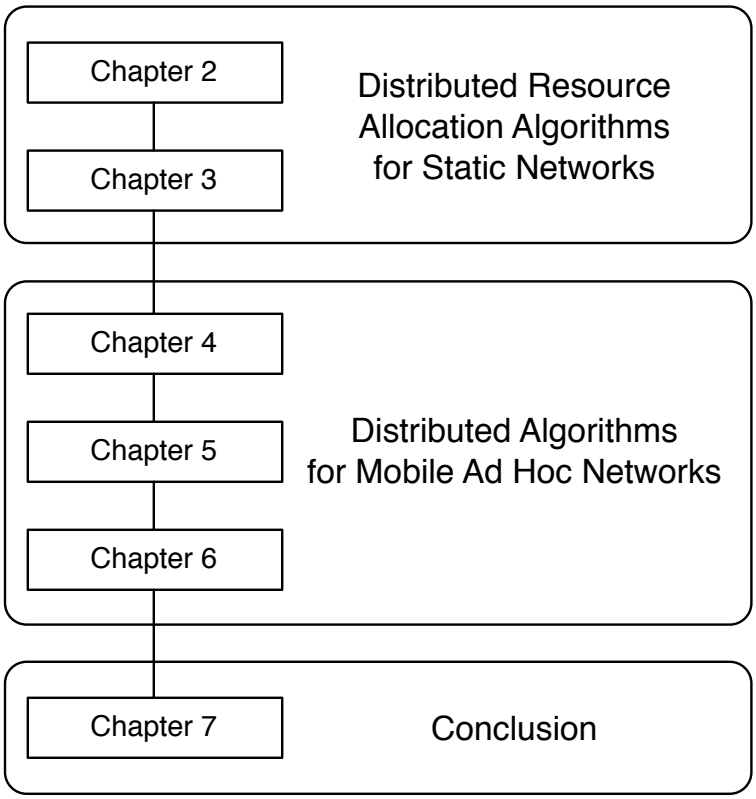


Figure 1.1: Dissertation organization: Sections 2 to 7.

1.2 Organization

Figure 1.1 shows how the remaining sections are organized. The first part of this dissertation considers distributed resource allocation algorithms for static networks and it consists of Sections 2 and 3. Sections 6, 4, and 5 constitutes the second part in which distributed algorithms for mobile ad hoc networks are dealt with. The outline of each section is as follows:

The dining philosophers problem with crash fault-tolerance and fairness in accessing the shared resource for message passing is considered in Section 2. The system model and problem specification are given in Section 2.2. Utilizing a certain failure

detector, a dining algorithm with failure locality 1 and fairness along with its proof of correctness is presented in Section 2.3. In Section 2.4, we implement and prove correct the failure detector that was originally used to solve our dining problem. Section 2.5 discusses how our result relates to other variations of dining problems and failure detectors.

Section 3 considers both crash and transient fault-tolerant dining for shared memory systems. Background and related work is provided in Section 3.2. The system model and problem specification is provided in Section 3.3. Again, utilizing failure detectors, a transient fault-tolerant failure-locality-1 dining algorithm is presented in Section 3.4. The correctness proof of our dining algorithm is also included in Section 3.4.

Maintaining neighbor knowledge of mobile nodes in a road network is considered in Section 4. Related work and the system model are presented in Sections 4.2 and 4.3, respectively. In Section 4.4, a deterministic collision-free broadcast schedule is presented. Section 4.5 explains how the parameters used in Section 4 relate to one another and then we show, in Section 4.6, that neighbor knowledge is maintained throughout the execution of our broadcast schedule under the assumption that initial neighbor knowledge is given. Given any two points A and B on the road network, a lower bound on the speed of message propagation from A to B is obtained in Section 4.7. In Section 4.8, we define clusters on the road network and show that, under certain conditions, neighbor knowledge is maintained when two clusters merge to form a new cluster. Section 4.9 discusses how initial neighbor knowledge can be obtained and presents practical values for the parameters.

Section 5 considers the problem of each mobile node detecting neighboring mobile nodes with which it can perform reliable point-to-point communication. Related work is presented in 5.2. The system model is given in Section 5.3 and the problem

specification is described in Section 5.4. Section 5.5 presents a periodic hello-based neighbor detection algorithm and its proof of correctness is given in Section 5.6. In Section 5.7, we apply parameter values to obtain the maximum distance between any two nodes that guarantees them to perform reliable point-to-point communication and compare the results of different parameter values.

Section 6 considers a problem of electing a leader among mobile nodes within a fixed region, called Regional Consecutive Leader Election (RCLE). In Section 6.2, we present related work that deals with leader election on mobile ad hoc networks. The system model and the RCLE problem definition is given in Section 6.3. We provide, in Section 6.4, a lower bound on the time it takes to elect a leader and subsequently, in Section 6.5, we provide and prove correct an asymptotically optimal RCLE algorithm. A condition on mobility, which guarantees the existence of a bounded communication diameter, is presented in Section 6.6.

Finally, Section 7 concludes this dissertation by summarizing contributions and providing future work.

2. DINING PHILOSOPHERS WITH BOUNDED WAITING AND FAILURE LOCALITY 1

In this section, we consider the dining philosophers problem, or simply dining, in an asynchronous message passing system where processes are subject to crash failures. Choy and Singh [14] showed that any asynchronous message-passing dining algorithm must have failure locality at least 2. There are several existing algorithms that solve dining with failure locality 2, showing that failure locality 2 is a tight bound [13, 67]. To circumvent this lower bound, previous work [68, 69, 62, 65, 63] has considered variations of dining along with utilizing failure detectors [11], system services that provide information about process crashes that need not always be correct.

Following this line of research, we present a specification for the BW-$\Box SX$-$FL1$ problem requiring the following (BW and $\Box SX$ stand for bounded waiting and perpetual strong exclusion, respectively, and failure locality 1 is abbreviated as $FL1$):

- *Exclusion:* No two neighbors (in the conflict graph) access their corresponding critical sections simultaneously.
- *FL1-progress:* A correct process that is trying to enter its critical section eventually does so if it only has correct neighbors and no correct process stays in its critical section forever.
- *FL1-BW:* If correct process p only has correct neighbors, then for any interval in which p is trying to enter its critical section, no neighbor of p enters its critical section more than a bounded[1] number of times.

[1]The bound may be different in different executions. See Section 2.5.

Recall from Section 1.1.1 that failure locality [13, 14] is a metric that corresponds to the maximum distance in the conflict graph between a crashed process p and any other process that is blocked from entering its critical section. Thus, $FL1$ indicates that each correct process that is at least two hops away from any crashed process is not blocked from entering its critical section.

We consider the local version of the well-known eventually perfect failure detector $(\Diamond P)^2$ [11] which repeatedly supplies each process p with a set of neighboring processes suspected to have crashed. We denote the local version of $\Diamond P$ as $\Diamond P^1$. Roughly, $\Diamond P^1$ has the following properties:

- *Completeness:* From some time on, for each correct process p, every crashed neighboring process of p is suspected by p.
- *Accuracy:* From some time on, for each correct process p, no correct neighboring process of p is suspected by p.

In this section, we consider that the output value of $\Diamond P^1$ is obtained via a query/response mechanism as in [37].

The Exclusion and $FL1$-progress properties of BW-$\Box SX$-$FL1$ prevent simultaneous access to the critical section and provide eventual guarantees in accessing the critical section, respectively. Exclusion and $FL1$-progress constitute the problem statement in [63]. In [63], the authors show that the $\Diamond P$ failure detector [11] is sufficient to solve their problem. They also show that $\Diamond P$ is a weakest[3] failure detector among the specific failure detectors introduced in [11] to solve their problem.

In addition to the Exclusion and $FL1$-progress properties, BW-$\Box SX$-$FL1$ includes an additional property. The $FL1$-BW property provides a level of fairness

[2]The specification of $\Diamond P$ states that, eventually, (1) every crashed process is suspected by every correct process, and (2) no correct process is suspected by any correct process.

[3]Weakest with respect to the relationship defined in [11] that is based on implementing one failure detector using another one.

in accessing the critical section between correct neighboring nodes ($FL1$-BW is a failure-locality-1 version of the bounded waiting property defined in [68, 69]). This is a desirable property for resource allocation when there is a need to prevent any process from dominating a shared resource by using it too frequently.

Finding a weakest failure detector for a problem is a way to characterize the minimum information regarding crash failures that is required to solve the problem. Mutual reducibility is a common proof technique used to identify a weakest failure detector. Suppose that, given a failure detector FD and a problem Q, we can solve Q using FD and we can also implement FD with a solution to Q (that is, Q and FD are mutually reducible). Now, for the sake of contradiction, suppose that there exists a failure detector FD' that is strictly weaker than FD (that is, FD' can be implemented using FD but it is impossible to implement FD using FD') and Q can be solved using FD'. Since FD' can be used to solve Q and a solution to Q can implement FD, by transitivity, FD' can be used to implement FD. This contradicts the assumption that FD' is strictly weaker than FD. So, FD is indeed a weakest failure detector for Q.

Now, a natural question arises: can we identify a weakest failure detector among all failure detectors for solving dining (or at least a variation of dining) with failure locality 1? We provide the first step towards identifying a weakest failure detector for BW-$\Box SX$-$FL1$ by showing that failure detector $\Diamond P^1$ and a solution to BW-$\Box SX$-$FL1$ are mutually reducible without preserving the underlying conflict graph topology. In our case, the conflict graph topology considered in one direction of the reduction is not necessarily the same in the other direction. Specifically, we first show that $\Diamond P^1$ is sufficient to solve BW-$\Box SX$-$FL1$ on any topology, and then we show the that using multiple instances of BW-$\Box SX$-$FL1$ on a "particular" (virtual) conflict graph, we can extract $\Diamond P^1$. This has the following implication: considering

an arbitrary conflict graph G and given $\Diamond P^1$, we can solve $BW\text{-}\Box SX\text{-}FL1$ on G but using "this" solution of $BW\text{-}\Box SX\text{-}FL1$ on G (the solution is tied with G), we may not be able to extract $\Diamond P^1$. Another implication is as follows: We have already shown that if a failure detector FD and a problem Q are mutually reducible, then FD is a weakest failure detector to solve Q. Considering a failure detector FD' that is strictly weaker than FD and can be used to solve Q, we have argued using transitivity through Q to show that FD is implementable using FD' which provided the desired contradiction. Now, if we substitute FD and Q with $\Diamond P^1$ and $BW\text{-}\Box SX\text{-}FL1$, respectively, then we might not be able to use transitivity through $BW\text{-}\Box SX\text{-}FL1$ to show that $\Diamond P^1$ is implementable using FD'. This is again because we can solve $BW\text{-}\Box SX\text{-}FL1$ using FD' on an arbitrary conflict graph G, however, using this solution which is tied with G, we might not be able to implement $\Diamond P^1$ (since G might not be the same as the particular graph used to extract $\Diamond P^1$). The above two implications in turn implies that, in order to conform with the "classical" notion of a weakest failure detector, mutual reducibility must preserve the underlying topology.

Previous work in [68, 69, 62, 65] considered the dining philosophers problem with *wait-freedom* and *eventual weak exclusion*. Wait-freedom corresponds to failure locality 0 and eventual weak exclusion states that *eventually*, no two live[4] neighbors access the critical section simultaneously. The difference between the work in [68, 69] and [62, 65] is that the former consider bounded waiting in accessing the critical section in the problem statement and the latter do not. Also, the results in [62, 65] show that $\Diamond P^1$ is a weakest failure detector which implies that wait-freedom and eventual weak exclusion themselves encapsulate bounded waiting, however, similar to our results, the results in [68, 69] show that $\Diamond P^1$ and a solution to the corresponding

[4]Live nodes are those nodes that have not yet crashed.

dining problem are mutually reducible without preserving the underlying topology.

We solve the BW-$\Box SX$-$FL1$ problem by carefully combining the asynchronous doorway technique in [68] and the concept of skepticism in [63]. For a process to enter the critical section, it must first enter the doorway by obtaining permission from all of its neighbors. Our solution satisfies the $FL1$-BW property by manipulating permission requests to enter the doorway. The concept of skepticism is used to satisfy all failure-locality-1 related properties: a process p becomes "skeptical" if *some* process in p's one-hop neighborhood is suspected by p's local failure detector and, as long as p is skeptical, p satisfies all requests from its neighbors.

In implementing $\Diamond P^1$, we use multiple instances of a solution to BW-$\Box SX$-$FL1$ as subroutines. In [69], the authors use a ping/ack protocol in conjunction with a dining subroutine for a correct process to directly detect crash failures; by direct detection, we mean that a correct node p detects a faulty neighbor q by directly exchanging messages with q. We use a similar ping/ack protocol to directly detect crash failures. However, there is a situation where direct detection is impossible. In that case, we use the ping/ack protocol to detect faulty processes by means of detecting starving[5] processes. Since our dining subroutine satisfies the $FL1$-progress property, the existence of a starving process corresponds to a crash failure in the neighborhood (see Section 2.4 for details).

2.1 Contributions

We consider a dining philosophers problem with perpetual strong exclusion. Since perpetual strong exclusion implies any other combination of eventual and/or weak exclusion, our algorithm can be used to satisfy weaker exclusion guarantees. We show that $\Diamond P^1$ is sufficient in solving the failure-locality-1 eventually bounded dining on

[5]Roughly, a correct process starves if it stops entering its critical section.

any topology. We also show that $\Diamond P^1$ can be implemented using multiple instances of BW-$\Box SX$-$FL1$ on a particular conflict graph. This serves as the first step towards identifying a weakest failure detector among all failure detectors for solving a dining problem with failure locality 1.

2.2 System Model and Problem Specification

Given a set Π of processes with unique ids and an undirected graph G with vertex set Π, we now define the BW-$\Box SX$-$FL1$ problem for G. The set of events X (for "exclusion") is defined to be

$$X \triangleq \{try_p,\ crit_p,\ exit_p,\ rem_p,\ crash_p : p \in \Pi\}.$$

Given a sequence σ over X, for any $p \in \Pi$, define p to be *faulty* in σ if $crash_p$ appears in σ and *correct* otherwise. For a finite sequence σ over X and any $p \in \Pi$, define $last(\sigma, p)$ to be the latest event in σ from $\{try_p,\ crit_p,\ exit_p,\ rem_p\}$. If $last(\sigma, p)$ is rem_p (resp. try_p, $crit_p$, $exit_p$), then we say that p is in its *remainder* (resp. *trying*, *critical*, *exiting*) section. We say that *Finite Eating holds* for a sequence σ over X if, for each correct $p \in \Pi$, $\sigma|p$ does not end with $crit_p$. Also, for a sequence σ over X, if $p \in \Pi$ is correct and all of p's neighbors in G are correct, then we say that p is *failure-insulated*. A sequence σ over X *satisfies the* BW-$\Box SX$-$FL1$ *specification for* G if it satisfies the following conditions:

- *Well-formedness:* For each $p \in \Pi$, if $\sigma|p$[6] is infinite, then it is the infinite repetition of try_p, $crit_p$, $exit_p$, rem_p; if $\sigma|p$ is finite, then it is either a prefix of this infinite repetition or it consists of a prefix of this infinite repetition followed by $crash_p$.
- *Exclusion:* For each prefix σ' of σ and each edge $\{p, q\}$ in G, if $last(\sigma', p) = crit_p$, then $last(\sigma', q) \neq crit_q$.

[6] $\sigma|p$ indicates the sequence of events in σ that occurred at p.

- *Finite Exiting:* For each correct $p \in \Pi$, $\sigma|p$ does not end with $exit_p$.
- *FL1-progress:* For each failure-insulated $p \in \Pi$, if Finite Eating holds for σ, then $\sigma|p$ does not end with try_p.
- *FL1-BW:* There exists an integer $k > 0$ such that for any two neighbors $p \in \Pi$ and $q \in \Pi$ where p is failure-insulated, every infix of σ that starts with try_p and ends with the next occurrence of $crit_p$ contains at most k occurrences of $crit_q$.

For any sequence σ over X, we say σ is *user-correct for* X if the shortest prefix of σ to violate Well-formedness (if any) ends in either rem_p or $crit_p$ for some $p \in \Pi$. (The intuition is that the user of the BW-$\Box SX$-$FL1$ module is not the first to violate Well-formedness; the user preserves Well-formedness)

Given a set Π of processes and an undirected graph G with vertex set Π, we now define the $\Diamond P^1$ failure detector problem for G. For each process $p \in \Pi$ on G, let N_p be the set of neighboring processes of p. The set of events D (for "detector") is defined to be

$$D \triangleq \{query_p,\ response(S)_p,\ crash_p :$$
$$p \in \Pi, S \subseteq N_p\}.$$

We use the same definitions of faulty and correct as above.

A sequence σ over D *satisfies the* $\Diamond P^1$ *specification* if it satisfies the following conditions:

- *Well-formedness:* For each $p \in \Pi$, if $\sigma|p$ is infinite, it consists of alternating $query_p$ and $response_p(.)$ events, starting with $query_p$; if $\sigma|p$ is finite, then it is either a prefix of such an infinite sequence or it consists of a prefix of such an infinite sequence followed by $crash_p$.
- *Liveness:* For each correct $p \in \Pi$, $\sigma|p$ does not end with $query_p$.

- *Completeness:* For each correct $p \in \Pi$, there is a suffix of σ in which the parameter S of each occurrence of $response(S)_p$ contains every faulty neighbor of p in G.
- *Accuracy:* For each correct $p \in \Pi$, there is a suffix of σ in which the parameter S of each occurrence of $response(S)_p$ does not contain any correct neighbor of p in G.

For any sequence σ over D, we say σ is *user-correct for* D if the shortest prefix of σ to violate Well-formedness (if any) ends in $response_p(.)$ for some $p \in \Pi$. (The intuition is that the user of the failure detector module is not the first to violate Well-formedness; the user preserves Well-formedness)

We model distributed algorithms as collections of state machines. Let Π be the set of nodes on which a distributed algorithm is running, with one state machine (process) per node. Each step of a process is triggered by an event and causes the process to change its local state. Events are partitioned into input and output events. An *input* event can occur at any time (i.e., it is always enabled), whereas an *output* event can only happen (be enabled) if certain preconditions are true in the state of the process.

We model a snapshot of the entire algorithm as a vector of states, one per process, which is called a *configuration.* In an *initial* configuration, each process is in an initial state. An *execution* of the algorithm is a sequence $C_0, e_1, C_1, e_2, C_2, \ldots$ of alternating configurations and events, beginning with an initial configuration, and if finite, ending with a configuration, that satisfies the following properties for each $i \geq 1$. Suppose event e_i occurs at process p.

- Event e_i is enabled in the state of p in the preceding configuration C_{i-1}.
- The only difference between C_{i-1} and C_i is that the state of p changes according to p's transition function (local algorithm).

Our algorithms execute in a system in which the processes communicate through an asynchronous message passing network, where the set of communication channels is described by an undirected graph G. If edge $\{p, q\}$ is in G, then there is a channel from p to q and a channel from q to p. The set of events for interacting with the message system is defined to be

$$N \triangleq \{send_p(m, q),\ recv_p(m, q),\ crash_p :$$
$$\{p, q\} \in G \text{ and } m \in M\},$$

where M is the set of messages. A sequence over N is said to *satisfy the message-passing specification* if it captures the properties that each channel is reliable (does not duplicate, alter, or inject messages, and does not lose messages sent to correct processes), asynchronous (unbounded delays), and FIFO (messages in each channel are delivered in the order in which they are sent). In addition, no $recv_p$, $send_p$, or $crash_p$ event occurs in the sequence after a $crash_p$ event occurs.

We next define what it means to be an *algorithm for the* BW-$\Box SX$-$FL1$ *problem for* G in a message-passing system augmented with a $\Diamond P^1$ failure detector. Note that the communication topology graph is the conflict graph. The set of input events is

$$\{try_p,\ exit_p,\ crash_p,\ response_p(S),\ recv_p(m, q) :$$
$$\{p, q\} \in G, S \subseteq N_p, m \in M\}.$$

The set of output events is

$$\{crit_p,\ rem_p,\ query_p,\ send_p(m, q) :$$
$$\{p, q\} \in G, m \in M\}.$$

Every execution E of the algorithm must satisfy the following property: If E satisfies the following three conditions:

1. $E|N$[7] satisfies the message-passing specification,
2. either $E|D$ is not user-correct for D or $E|D$ satisfies the $\Diamond P^1$ specification,
3. $E|X$ is user-correct for X,

then $E|X$ satisfies the $BW\text{-}\Box SX\text{-}FL1$ specification.

Now we define what it means to be an *algorithm for the $\Diamond P^1$ problem for G* in a message-passing system augmented with $BW\text{-}\Box SX\text{-}FL1$ subroutines. The algorithm uses some number, say H, of instances of a solution to the $BW\text{-}\Box SX\text{-}FL1$ problem. The h-th instance is for some undirected graph G^h with vertex set Θ^h where there exists a function f that maps Θ^h to Π; $\exists f : \Theta^h \rightarrow \Pi$. For the h-th instance, we denote the set of events by $X^h = \{try_s,\ crit_s,\ exit_s,\ rem_s,\ crash_p : s \in \Theta^h, p = f(s)\}$. We note that for each $s \in \Theta^h$ where $f(s) = p$, $crash_p$ is a "synonym" for $crash_s$ with respect to the formal definition of the $BW\text{-}\Box SX\text{-}FL1$ problem.

The set of input events for the $\Diamond P^1$ algorithm is

$$\bigcup_{h=1}^{H}\{crit_s,\ rem_s : s \in \Theta^h\} \cup \{crash_p,\ query_p,\ recv_p(m,q) : \{p,q\} \in G, m \in M\}.$$

The set of output events is

$$\bigcup_{h=1}^{H}\{try_s,\ exit_s : s \in \Theta^h\} \cup \{response_p(S),$$
$$send_p(m,q) : S \subseteq N_p, \{p,q\} \in G, m \in M\}.$$

Every execution E of the algorithm must satisfy the following property: If E satisfies the following three conditions:

1. $E|N$ satisfies the message-passing specification,
2. for each h, $1 \leq h \leq H$, either $E|X^h$ is not user-correct for X^h or $E|X^h$ satisfies the $BW\text{-}\Box SX\text{-}FL1$ specification for G^h,
3. $E|D$ is user-correct for D,

then $E|D$ satisfies the $\Diamond P^1$ specification.

[7] $E|N$ indicates the sequence of events in E with respect to set N.

2.3 Solving BW-$\Box SX$-$FL1$ using $\Diamond P^1$

The algorithm in Figures 2.1, 2.2, and 2.3 solves the problem of BW-$\Box SX$-$FL1$ using a solution to the $\Diamond P^1$ problem. Roughly speaking, this algorithm uses a ping/ack protocol in conjunction with asynchronous doorways [13] to provide fairness and uses a fork/request protocol to provide safety in entering a critical section. A node has to first enter the doorway for it to enter its critical section.

The pseudocode in Figures 2.1, 2.2, 2.3, and the figures found on p. 45-48 uses the following conventions:

- An *immediate* occurrence of an output event e is indicated as "generate e" in the code.
- When node p crashes, we assume that p simply stops its execution.
- We assume that each action block is executed atomically.

2.3.1 Algorithm Overview

We first consider the case when a correct node $p \in \Pi$ does not have any faulty neighbors and $\Diamond P^1$ has stabilized such that $S = \emptyset$ for all events $response_p(S)$ thereafter. In this case, $skeptical = F$ holds; this indicates that p does not suspect any of its neighbors to be faulty. When p enters a trying section, it sets variable *dining* to *try* and sends ping messages to all its neighbors; variable *dining* keeps track of the most recent occurrence of *try* and *crit* events. Node p then waits until it receives ack messages from all of its neighbors to enter the doorway; the value of $ack[q]$ tells whether node p has received an ack message from its neighbor q and variable *inside* is used to indicate whether a node is inside the doorway.

Upon receiving a ping message from a neighbor q, node p does *not* send an ack message back to q if (1) p is inside the doorway, or (2) p already received a ping message from q in the current trying section; otherwise p immediately sends an ack

```
    ⟨Variables and Initialization⟩
 1: N_p; // neighbor set of p
 2: ∀q ∈ N_p : reqToken[q]; // initialized as in Section 2.3.1
 3: ∀q ∈ N_p : fork[q]; // initialized as in Section 2.3.1
 4: dining ← rem; // dining state of p
 5: inside ← F;
 6: skeptical ← F;
 7: ∀q ∈ N_p : ack[q] ← F;
 8: ∀q ∈ N_p : replied[q] ← F;
 9: ∀q ∈ N_p : deferred[q] ← F;
10: generate query_p;
    ---------------------------------------------------------------

11: ⟨When try_p occurs⟩
12: dining ← try;
13: for all q ∈ N_p do
14:     generate send_p(⟨ping⟩, q);

15: ⟨When recv_p(⟨ping⟩, q) occurs⟩
16: if (inside ∨ replied[q]) ∧ ¬skeptical then
17:     deferred[q] ← T;
18: else
19:     send ack to q;
20:     if dining = try then
21:         replied[q] ← T;

22: ⟨When recv_p(⟨ack⟩, q) occurs⟩
23: ack[q] ← T;
24: if (∀r ∈ N_p : ack[r]) ∧ (dining = try) ∧ ¬skeptical then
25:     EnterDoorway();
```

Figure 2.1: Solving BW-$\Box SX$-$FL1$ with $\Diamond P^1$; code for node p (part 1 of 3).

```
26: ⟨When recv_p(⟨request⟩, q) occurs⟩
27: reqToken[q] ← T;
28: if ¬inside ∨ ((dining = try) ∧ (p < q)) then
29:     generate send_p(⟨fork⟩, q);
30:     fork[q] ← F;
31:     if inside then
32:         generate send_p(⟨request⟩, q);
33:         reqToken[q] ← F;

34: ⟨When recv_p(⟨fork⟩, q) occurs⟩
35: fork[q] ← T;
36: if (dining = try) ∧ inside then
37:     AllForks();

38: ⟨When exit_p occurs⟩
39: dining ← exit;
40: SatisfyRequests();
41: generate rem_p;

42: ⟨When response_p(S) occurs⟩
43: if S ≠ ∅ then
44:     skeptical ← T;
45:     if dining ≠ crit then
46:         SatisfyRequests();
47: else
48:     skeptical ← F;
49:     if (dining = try) ∧ ¬inside ∧ (∀q ∈ N_p : ack[q]) then
50:         EnterDoorway();
51: generate query_p;
```

Figure 2.2: Solving BW-$\Box SX$-$FL1$ with $\Diamond P^1$; code for node p (part 2 of 3).

```
52: procedure AllForks()
53:    if ∀q ∈ N_p : fork[q] then
54:       dining ← crit;
55:       generate crit_p;

56: procedure EnterDoorway()
57:    inside ← T;
58:    for all q ∈ N_p do
59:       ack[q] ← F;
60:       replied[q] ← F;
61:       if (reqToken[q] ∧ ¬fork[q]) then
62:          generate send_p(⟨request⟩, q);
63:          reqToken[q] ← F;
64:    AllForks();

65: procedure SatisfyRequests()
66:    inside ← F;
67:    for all q ∈ N_p where (reqToken[q] ∧ fork[q]) do
68:       generate send_p(⟨fork⟩, q);
69:       fork[q] ← F;
70:    for all q ∈ N_p where deferred[q] do
71:       generate send_p(⟨ack⟩, q);
72:       deferred[q] ← F;
```

Figure 2.3: Solving BW-$\Box SX$-$FL1$ with $\Diamond P^1$; code for node p (part 3 of 3).

message to q. Arrays $replied[\cdot]$ and $deferred[\cdot]$ handle this action of deferring an ack request which results in providing the $FL1$-BW property: no neighbor of p enters a critical section more than *two* times while p is continuously in a trying section.

Node p uses array $fork[\cdot]$ to determine whether it can generate event $crit_p$ (enter the critical section). Array $reqToken[\cdot]$ is used to keep track of *request* messages exchanged between p and its neighbors. For the algorithm in Figures 2.1, 2.2, and 2.3, we assume that arrays $reqToken[\cdot]$ and $fork[\cdot]$ are initialized as follows: for each pair of neighboring nodes $p \in \Pi$ and $q \in \Pi$, $reqToken[p] = T$ if and only if

$fork[q] = F$. Node p sets $fork[q]$ to T when it receives a $fork$ message and $fork[q]$ to F when it sends a $fork$ message. Similarly, node p sets $reqToken[q]$ to T when it receives a *request* message and $reqToken[q]$ to F when it sends a *request* message.

For two neighboring nodes p and q, there exists a unique fork token (or simply fork) and a unique request token shared between p and q such that $fork[q] = T$ at p (resp. $fork[p] = T$ at q) corresponds to node p (resp. q) *holding* the fork and $reqToken[q] = T$ at p (resp. $reqToken[p] = T$ at q) corresponds to node p (resp. q) *holding* the request token.

When node p enters the doorway, p requests for the missing forks by sending request messages to its neighbors and if p already holds all the forks, then p enters its critical section. When p receives a request message from its neighbor q, p determines whether it can immediately send the fork to q; node p sends the fork to q if (1) p is outside the doorway, or (2) p is in its trying section and the id of q is greater than p's id. In the case when a fork request from node q is deferred at node p, the value of $reqToken[q]$ will remain as T until node p exits the critical section.

At node p, whenever event $response_p(S)$ occurs with $S = \emptyset$, it checks the suitability of entering the doorway; if p is in its trying section, outside of the doorway, and received ack messages from all of its neighbors, then p enters the doorway.

When node p is in a trying section, inside the doorway, and holds all forks, then p enters a critical section. When event $exit_p$ occurs at node p (when p enters an exiting section), node p goes outside of the doorway, sends forks to those neighbors that sent a request message to it, satisfies all deferred ack requests, and enters its remainder section. This fork/request protocol basically provides the *Exclusion* property. Also, the combination of the ping/ack protocol and the fork/request protocol provides property $FL1$-*progress*.

Now, we consider the case when a correct node $p \in \Pi$ has a faulty neighbor and

$\Diamond P^1$ has stabilized such that $S \neq \emptyset$ for all events $response_p(S)$ thereafter. In this case, $skeptical = T$ holds; this indicates that p suspects it has a faulty neighbor. First note that whenever event $response_p(S)$ occurs with $S \neq \emptyset$ at node p, as long as p is not in a critical section, p exits the doorway and satisfies all deferred fork and ack requests. Upon receiving a ping message sent by a neighboring node q, node p will immediately send back an ack message to q. Also, since $skeptical = T$ holds, node p will *not* enter the doorway even though it is in its trying section and received ack messages from all of its neighbors. Note that p does not need to enter its critical section if it has a crashed neighbor.

2.3.2 Proof Outline

In this section, we only provide an outline of the proof of the algorithm in Figures 2.1 2.2, and 2.3.

The Finite Exiting property directly follows from the pseudocode. Using the fact that the variable *dining* correctly keeps track of the most recent occurrence of *try* and *crit* events (Lemma 2.3.2), we show that *crit* and *rem* events only happen after *try* and *exit*, respectively, which proves the Well-formedness property (Lemma 2.3.3). To prove the Exclusion property (Lemma 2.3.7), we first identify invariants that are true in every configuration of any execution (Lemma 2.3.6). The key invariants used in showing the Exclusion property are: (a) the fork shared between any two neighbors is unique, and (b) if a node is in its critical section, then it is inside the doorway and holds all forks shared between itself and all of its neighbors.

The $FL1$-progress property (Lemma 2.3.14) is shown in two steps. We first prove that for each correct node p that only has correct neighbors, if p is inside the doorway, then it eventually enters its critical section. Then, we show that if p is outside the doorway, then it eventually enters the doorway. Both steps rely on the assumption

that no correct process stays in its critical section forever and have a similar proof structure: We first show that if there exists a node that is not making progress, then there is an array variable that stops changing its values (Lemmas 2.3.10 and 2.3.12). We then show by means of contradiction that the array variable cannot be stabilized, due to the total ordering of node ids and the total ordering of events in an execution (Lemmas 2.3.11 and 2.3.13).

The following facts are used to show that the $FL1$-BW property holds (Lemma 2.3.15): Suppose node p is correct and only has correct neighbors. While node p is continuously in a trying section and outside the doorway, the use of array variable $replied[\cdot]$ shows that p does not send an ack message more than one time to any neighboring node. Also, notice that if p is inside the doorway, then it does not send ack messages to its neighbors.

2.3.3 Proof of Correctness

Let E be any execution of the algorithm in Figures 2.1, 2.2, and 2.3 such that $E|N$ satisfies the message-passing specification, either $E|D$ is not user-correct for D or $E|D$ satisfies the $\Diamond P^1$ specification, and $E|X$ is user-correct for X.

After the initial generation of output event $query_p$ (line 10), subsequent $query_p$ events are generated only after event $response_p(b)$ is enabled (line 51). Hence, $E|D$ is user-correct and we may assume that $E|D$ satisfies the $\Diamond P^1$ specification.

2.3.3.1 Finite Exiting, Well-formedness, and Exclusion

Lemma 2.3.1. *$E|X$ satisfies Finite Exiting.*

Proof. This follows directly from the code (lines 38-41). □

Lemma 2.3.2. *For every $p \in \Pi$ and every prefix E' of E, if $dining_p = try$, then $last(E'|X, p) = try_p$, and if $dining_p = crit$, then $last(E'|X, p) = crit_p$.*

Proof. Initially $dining_p = rem$, so the lemma is vacuously true. When try_p occurs, $dining_p$ is set to try. When $crit_p$ occurs, $dining_p$ is set to $crit$. The only other time when $dining_p$ changes is when $exit_p$ occurs, in which case $dining_p$ is set to $exit$. □

Lemma 2.3.3. *$E|X$ satisfies Well-formedness.*

Proof. Suppose in contradiction $E|X$ does not satisfy Well-formedness. By the assumption that $E|X$ is user-correct for X, the first error is because the algorithm outputs $crit_p$ or rem_p at a wrong time. However, rem_p is only generated immediately after $exit_p$ occurs (cf. lines 38–41), which is correct.

But p only generates $crit_p$ in AllForks, which is called in two places, line 37 and line 64. Line 64 is part of EnterDoorway, which is called in two places, line 25 and line 50. In all cases, we see from the code that $dining_p = try$ (lines 36, 24, 49) when $crit_p$ is generated. By Lemma 2.3.2, the most recent preceding event in X is try_p, which is correct, a contradiction.

Thus $E|X$ satisfies Well-formedness. □

Definition 2.3.4. *If $fork_p[q] = T$, then we say p* has a $\{p, q\}$-fork*; if $fork_q[p] = T$, then we say q* has a $\{p, q\}$-fork*; if a $fork$ message is in transit from p to q, then we say* the channel from p to q has a $\{p, q\}$-fork*; and if a $fork$ message is in transit from q to p, then we say* the channel from q to p has a $\{p, q\}$-fork.

Definition 2.3.5. *If $reqToken_p[q] = T$, then we say p* has a $\{p, q\}$-request*; if $reqToken_q[p] = T$, then we say q* has a $\{p, q\}$-request*; if a request message is in transit from p to q, then we say* the channel from p to q has a $\{p, q\}$-request*; and if a request message is in transit from q to p, then we say* the channel from q to p has a $\{p, q\}$-request.

Lemma 2.3.6. *The following are invariants (true in every configuration of E) for all neighbors p and q.*

(A) There is exactly one $\{p, q\}$-fork.

(B) If $dining_p = crit$, then $fork_p[q] = T$ and $inside_p = T$.

(C) If a request message is in transit from p to q, then either a $fork$ message precedes the request message in the channel or q has the $\{p, q\}$-fork.

(D) If $reqToken_p[q] = T$, then a $fork$ message is not in transit from q to p.

(E) There is exactly one $\{p, q\}$-request.

(Note: (B) can be proved independently, as can (E). (A), (C) and (D) need to be proved all together, and they rely on (E).)

Proof. By induction on the configurations in E, which we denote $C_0, C_1, \ldots$.

–(**Base case**) By the initialization, (A) through (E) are true in C_0.

–(**Inductive case**) Suppose (A)-(E) are true in configuration C_{t-1} and show they are true in configuration C_t. We consider every possibility for the event taking the system from C_{t-1} to C_t. When try_p and $recv_p(ping, q)$ occur, no changes are made that affect the truth of any of the predicates.

Case 1: The event is $recv_p(ack, q)$.

(A) No changes affect $fork$ variables or $fork$ messages. By the inductive hypothesis, (A) is true in C_{t-1} and thus it remains true in C_t.

(B) The only change that can affect (B) is if $dining_p$ is set to *crit* in line 54 of AllForks, which is called from EnterDoorway. The check in line 53 of AllForks ensures that $fork_p[q]$ is true, and the assignment in line 56 of enterDoorway sets $inside_p$ to true.

(C) The only change that can affect (C) is if p sends a *request* message to q in line 62 of EnterDoorway. We must show that either a $fork$ message precedes the *request* message in the channel from p to q or q has the $\{p, q\}$-fork. Since $reqToken_p[q] = T$ in C_{t-1}, the inductive hypothesis (D) implies that a $fork$ message is not in transit from q to p. Since $fork_p[q] = F$ in C_{t-1}, the inductive hypothesis (A) implies that either a $fork$ message is in transit from p to q or q has the fork in C_{t-1}. Thus (C) is true in C_t.

(D) The only change relevant to (D) is if $reqToken_p[q]$ is set to false in line 63 of EnterDoorway. However, in this case (D) is vacuously true in C_t.

(E) The only change relevant to (E) is if lines 61–63 are executed in EnterDoorway and p sends a request message to q. In this case, the $\{p, q\}$-request moves from being (uniquely) at p to being (uniquely) in transit from p to q. Uniqueness follows from the inductive hypothesis (E).

Case 2: The event is $recv_p(request, q)$.

(A) In C_{t-1}, a *request* message is at the head of the channel from q to p (since channels are FIFO). By the inductive hypothesis (C), $fork_p[q] = T$ in C_{t-1}. By the inductive hypothesis (A), in C_{t-1}, no $fork$ message is in transit from p to q or from q to p, and $fork_q[p] = F$. By the code, in C_t, either a $fork$ message is in transit from p to q and $fork_p[q] = F$, or no $fork$ message is in transit from p to q and $fork_p[q] = T$. Thus (A) still holds.

(B) The only change that could affect (B) is setting $fork_p[q]$ to false in line 30. But this only happens if $inside_p$ is false (cf. line 28). By the inductive hypothesis (B), $dining_p \neq crit$ in C_{t-1}. Thus $dining_p$ is still not equal to *crit* in C_t and the change to $fork_p[q]$ does not invalidate (B) in C_t.

(C) If p sends a *request* message to q in line 32, it previously sends a $fork$ message to q in line 29.

(D) In C_{t-1}, a *request* message is at the head of the channel from q to p. Thus by the inductive hypothesis (C), p has the $\{p, q\}$-fork and no $fork$ message is in transit between p and q in either direction in C_{t-1}. If $reqToken_p[q] = T$ in C_t (i.e., line 33 is not executed), then there is still no $fork$ message in transit from q to p.

Suppose p sends the fork to q. Then we must show that $reqToken_q[p] = F$. This follows from the inductive hypothesis (E): since the *request* message is in transit from q to p in C_{t-1}, it must be that $reqToken_q[p] = F$.

(E) If lines 32–33 are not executed, then the $\{p, q\}$-request goes from being (uniquely) in transit from q to p to being (uniquely) at p. If lines 32–33 are executed, then the $\{p, q\}$-request goes from being (uniquely) in transit from q to p to being (uniquely) in transit from p to q (The uniqueness follows from the inductive hypothesis (E)).

Case 3: The event is $recv_p(fork, q)$.

(A) In C_{t-1}, a $fork$ message is at the head of the channel from q to p. By the inductive hypothesis (A), in C_{t-1}, no $fork$ message is in transit from p to q, and $fork_p[q]$ and $fork_q[p]$ are both false. By the code, in C_t, no $fork$ message is in transit from q to p (since it is removed from the channel in order to be received) and $fork_p[q] = T$.

(B) The only change that can affect (B) is if $dining_p$ is set to *crit* in line 54 of AllForks, which is called from line 37. The check in line 53 of AllForks ensures that $fork_p[q]$ is true, and the check in line 36 ensures that $inside_p$ is true.

(C) No change affects the validity of (C) in C_t.

(D) No change affects the validity of (D) in C_t.

(E) No change affects the validity of (E) in C_t.

Case 4: The event is $exit_p$.

(A) The only changes that possibly affect (A) occur if lines 67–69 in SatisfyRequests are executed. In this case, the fork changes from being (uniquely) at p to being (uniquely) in transit from p to q. (The uniqueness is due to the inductive hypothesis (A).)

(B) Since $dining_p$ is set to a value other than $crit$, (B) is vacuously true in C_t.

(C) The only change that possibly affects (C) occurs if p sends a $fork$ message to q in line 68 of SatisfyRequests. We must show that no $request$ message is in transit from q to p. By the code, $reqToken_p[q] = T$ in C_{t-1} (cf. line 67). By the inductive hypothesis (E), there is no $request$ message in transit.

(D) The only change that possibly affects (D) occurs if p sends a $fork$ message to q. We must show that $reqToken_q[p] = F$. In C_{t-1}, if p sends a $fork$ message to q, it must be that $reqToken_p[q] = T$. So by the inductive hypothesis (E), $reqToken_q[p] = F$.

(E) No change affects the validity of (E) in C_t.

Case 5: The event is $response_p(S)$.

(A) The only changes that possibly affect (A) are if lines 67–69 in SatisfyRequests are executed. The same argument as for $exit_p$ holds.

(B) Suppose $S \neq \emptyset$. Then changes relevant to (B) are only made if $dining_p$ is not *crit* (cf. line 45), and so (B) is vacuously true in C_t.

Suppose $S = \emptyset$. The only change that can affect (B) is if $dining_p$ is set to *crit* in line 54 of AllForks, which is called from EnterDoorway. The check in line 53 of AllForks ensures that $fork_p[q]$ is true, and the assignment in line 57 of EnterDoorway sets $inside_p$ to true.

(C) Suppose $S \neq \emptyset$. The only change relevant to (C) is if p sends a $fork$ message to q in line 68 of SatisfyRequests. We must show that no *request* message is in transit from q to p. By the code, $reqToken_p[q] = T$ in C_{t-1} (cf. line 67). By the inductive hypothesis (E), there is no *request* message in transit.

Suppose $S = \emptyset$. The only change relevant to (C) is if p sends a *request* message to q in line 62 of EnterDoorway. We must show that either a $fork$ message precedes the *request* message in the channel from p to q, or q has the $\{p, q\}$-fork. Since $reqToken_p[q] = T$ in C_{t-1}, the inductive hypothesis (D) implies that a $fork$ message is not in transit from q to p. Since $fork_p[q] = F$ in C_{t-1}, the inductive hypothesis (A) implies that either a $fork$ message is in transit from p to q, or q has the $\{p, q\}$-fork in C_{t-1}. Thus (C) is true in C_t.

(D) Suppose $S \neq \emptyset$. The only change relevant to (D) is if p sends a $fork$ message to q in line 68 of SatisfyRequests. We must show that $reqToken_q[p] = F$. In C_{t-1}, if p sends a $fork$ message to q, it must be that $reqToken_p[q] = T$. So by the inductive hypothesis (E), $reqToken_q[p] = F$.

Suppose $S = \emptyset$. The only change relevant to (D) is if $reqToken_p[q]$ is set to false in line 63 of EnterDoorway. However, in this case (D) is vacuously true in C_t.

(E) Suppose $S \neq \emptyset$. Then no change relevant to (E) is made in C_t.

Suppose $S = \emptyset$. The only change relevant to (E) occurs if lines 62–63 are executed in EnterDoorway and p sends a *request* message to q. In this case, the $\{p, q\}$-request moves from being (uniquely) at p to being (uniquely) in transit from p to q (The uniqueness follows from the inductive hypothesis (E)).

Therefore, (A), (B), (C), (D), and (E) are invariants for all neighbors p and q. □

Lemma 2.3.7. *$E|X$ satisfies Exclusion.*

Proof. Suppose p and q are two processes that are neighbors in the conflict graph and E' is a prefix of E such that $last((E'|X), p)$ is $crit_p$. Consider the configuration at the end of E'. By Lemma 2.3.2, $dining_p = crit$. By Lemma 2.3.6(B), $fork_p[q] = T$. By Lemma 2.3.6(A), $fork_q[p] = F$. By Lemma 2.3.6(B), $dining_q \neq crit$. Thus, by Lemma 2.3.2, Lemma 2.3.3, and the fact that $dining_p = exit$ when $exit_p$ occurs, $last(E'|X, q)$ is not $crit_q$. □

2.3.3.2 FL1-progress

Once a node enters a trying section, that node must (1) acquire acks from all of its neighbors to enter the doorway and (2) when inside the doorway, acquire all forks shared between itself and all of its neighbors to enter its critical section. We first show that for each correct node p, if (a) p is in its trying section, (b) p is inside the doorway, (c) all of p's neighbors are correct, and (d) all correct neighbors of p eventually exits their critical section, then p eventually enters its critical section.

Lemma 2.3.8. *For all nodes $p \in \Pi$, when p enters a trying section, $inside = F$ holds.*

Proof. Note that initially $inside = F$ at p. Since user-correctness of $E|X$ is assumed (try_p is the first event in X that occurs at p), the first time p enters a trying section,

$inside = F$ holds. By the Well-formedness property (Lemma 2.3.3),for all future occurrences of event try_p, there exists a preceding occurrence of event $exit_p$ which calls SatisfyRequests() and sets $inside$ to F. Since the value of $inside$ can only be modified to T by calling EnterDoorway() and EnterDoorway() can be called only when p is in a trying section (Lemma 2.3.2 shows that $dining = trying$ at p implies that p is in a trying section), $inside = F$ holds when p enters a trying section. □

For all nodes $p \in \Pi$ and $q \in N_p$, we say that an ack message $\langle ack \rangle$ sent from q to p is a *consequence* of a ping message $\langle ping \rangle$ sent from p to q if (1) q sends $\langle ack \rangle$ by receiving $\langle ping \rangle$ from p (line 19) or (2) $deferred[p]$ is modified from F to T at q by receiving $\langle ping \rangle$ from p which enabled q to send $\langle ack \rangle$ by executing line 71.

Lemma 2.3.9. *For all nodes $p \in \Pi$ and $q \in N_p$ and for all occurrences of send events $send_p(\langle ping \rangle, q)$, (1) the next successive send event $send_p(\langle ping \rangle', q)$ (if any) occurs only after an occurrence of $recv_p(\langle ack \rangle, q)$ from q at some point after the occurrenct of $send_p(\langle ping \rangle, q)$, and (2) this ack message $\langle ack \rangle$ is a consequence of $\langle ping \rangle$ and it is the only ack message that p receives in between the transmissions of $\langle ping \rangle$ and $\langle ping \rangle'$.*

Proof. We first prove part (1). Node p only sends a ping message to q when p enters a trying section (line 14). Also, when p enters a trying section, $inside = F$ holds (Lemma 2.3.8). By the Well-formedness property (Lemma 2.3.3), between any two trying sections, node p must enter a critical section. This implies that there exists a call to EnterDoorway() between any two ping message transmissions. The reason for this is that for p to enter a critical section, it must call AllForks(), and for AllForks() to be called, $inside = T$ must hold (the only way to set $inside = T$ is be calling EnterDoorway; line 57). For EnterDoorway() to be called, p must gather

ack messages from all of its neighbors (lines 24 and 49) which further implies that p must receive an ack message from q.

For part (2), note that by receiving one ping message from p, exactly one ack message can be generated at q. This is because executing line 19 at q is a direct result of receiving a ping message from p and executing line 71 at q is a result of $deferred[p]$ being T which is set by receiving a ping message from p (and after executing line 71, $deferred[p]$ is reset to F). Applying part (1) proves the lemma. □

We next show that if a failure-insulated node $p \in \Pi$ does not enter its critical section even though it is in a trying section and inside the doorway, then the values in array $fork[\cdot]$ eventually stop changing.

Lemma 2.3.10. *Suppose that there exists a failure-insulated node $p \in \Pi$. Also, suppose that there exists a suffix E_1 of E that begins with configuration $C_{t'}$ such that p never enters its critical section even if p is in a trying section and $inside = T$ holds at p. Then, there exists a suffix of E in which for all nodes $q \in N_p$, $fork[q]$ stops changing at p.*

Proof. Since $E|D$ satisfies the $\Diamond P^1$ specification, there exists a suffix E_2 of E where for all nodes $r \in \Pi$, every occurrence of event $response_r(S)$ returns $S \neq \emptyset$ if and only if r has a faulty neighbor. For our proof, we consider the suffix $E_1 \cap E_2$[8] of E.

First note that whenever node p sends a fork to its neighbor q, $fork_p[q]$ is set to F and the only case that $fork_p[q]$ is set to T is when p receives a fork message from q. If node p ever sends a fork message to its neighbor q, and if p ever receives a fork message back from q (note that the fork is unique by Lemma 2.3.6(A)), then there are four cases that node q could have sent the fork message back to p:

[8]For two suffixes E_1 and E_2 of an execution E, $E_1 \cap E_2$ corresponds to E_1 (resp. E_2) if E_1 (resp. E_2) begins later than E_2 (resp. E_1). If E_1 and E_2 are equivalent, then $E_1 \cap E_2$ corresponds to E_1.

–(*Case* 1) q, which has a faulty neighbor, sent the fork by calling SatisfyRequests() on line 46: In this case, $skeptical = T$ holds at q forever. As a result, if q is ever in a trying section, then EnterDoorway() is never called and thus $inside = F$ holds thereafter. Hence, q cannot send a request message to p which proves that there exists a suffix of E in which $fork_p[q]$ stops changing.

–(*Case* 2) q, which is failure-insulated, sent the fork message by calling SatisfyRequests() on line 40: Since we only consider suffix $E_1 \cap E_2$ of E, SatisfyRequests() can only be called at q when it enters an exiting section; every occurrence of $response_q(S)$ in $E_1 \cap E_2$ returns $S = \emptyset$. At the time when q sends the fork message, it must hold the request token (line 67). For node q to reacquire the fork it must send a request message to p by calling EnterDoorway(). Lemma 2.3.9 tells us that for node q to call EnterDoorway(), it must send ping messages to all of its neighbors (by entering a trying section) and then gather ack messages from all of its neighbors that are consequences of those ping messages. However, since $inside = T$ holds at node p, the ping message from q will be deferred (line 16) and since p never enters its critical section, the Well-formedness property (Lemma 2.3.3) shows that p never calls SatisfyRequests(). Note that, by Lemma 2.3.9, p can only send an ack message back to q by calling SatisfyRequests(). Hence, for Case 2, there exists a suffix of E in which $fork_p[q]$ stops changing.

–(*Case* 3) q, which is failure-insulated, sent the fork message (line 29) when $inside = F$ holds: First note that the fork shared between p and q is unique by Lemma 2.3.6(A). We further divide into two cases:

—(*Case* 3a) Suppose $p > q$: If q ever calls EnterDoorway() and sets $inside$ to T (for q to send a request message to p, $inside = T$ must hold; see lines 31 and 57), and if p ever receives a request message from q, then p defers the fork request due to the condition on line 28. The only case that node p can send the fork is

by calling SatisfyReqeust() which can only be called by entering an exiting section (since p is failure-insulated, in $E_1 \cap E_2$, every occurrence of event $response_p(S)$ returns $S = \emptyset$ which prevents the call to SatisfyRequests() at line 46). Since p never enters a critical section by assumption, p never enters an exiting section by the Well-formedness property (Lemma 2.3.3). Hence, $fork_p[q] = T$ holds in $E_1 \cap E_2$.

—(*Case* 3b) Suppose $p < q$. If q ever calls EnterDoorway() and sets *inside* to T and if p ever receives a request message from q, then, different from Case 3a, p sends the fork to q and then p immediately sends a request message to q (by FIFO message delivery, q will receive the fork prior to receiving the request message). For q to send back the fork to p, *inside* must to be set to F by calling SatisfyRequests() which can only be done by entering an exiting section. This is because q being failure-insulated ensures that, in $E_1 \cap E_2$, every occurrence of event $response_q(S)$ returns $S = \emptyset$ which in turn prevents the call to SatisfyRequests() at line 46. The rest of the proof is similar to Case 2.

–(*Case* 4) q, which only has correct neighbors, sent the fork message (line 29) when $p > q$ and $inside = T$ holds while q is in a trying section: The proof is similar to Case 3a. □

Lemma 2.3.11. *Consider each failure-insulated node $p \in \Pi$. If (1) Finite Eating holds for $E|X$, (2) p is in a trying section, and (3) inside $= T$ holds at p, then p eventually enters its critical section.*

Proof. Suppose, in contradiction, that there exists a non-empty set $U \subseteq \Pi$ of failure-insulated nodes such that for all nodes $u \in U$, u is in its trying section and $inside = T$ holds at u but never enters its critical section. Then, by Lemma 2.3.10, there exists a suffix E_1 of E in which for all nodes in U, array $fork[\cdot]$ stops changing.

Since $E|D$ satisfies the $\Diamond P^1$ specification, there exists a suffix E_2 of E in which

for all nodes $r \in \Pi$, every occurrence of event $response_r(S)$ returns $S \neq \emptyset$ if and only if r has a faulty neighbor. We only consider suffix $E_1 \cap E_2$ of E for our proof.

Since a node enters a critical section if it is in a trying section, inside the doorway, and holds all of the forks shared between itself and its neighbors, for each node $u \in U$, there exists a node $q \in N_u$ such that $fork[u] = T$ holds at q. We first show that q is in U and $q > u$. Suppose not. First note that since $inside = T$ and $fork[q] = F$ holds at u, node u sends a request message to q. We consider four cases: If q has a faulty neighbor, then eventually SatisfyRequests() is called (line 46) which enables a fork message to be sent from q to u after q receives the request message. If q is failure-insulated and if q ever enters its critical section or exiting section, the Well-formedness property ensures that eventually SatisfyRequests() is called (line 40) which enables a fork to be sent from q to u after q receives the request message. If q is failure-insulated and if $inside = F$ holds at q, then upon receiving a request message from u, q sends the fork to u (line 29). Finally, if q is failure-insulated, q is in its trying section, $inside = T$ holds at q, and $q < u$, then upon receiving a request messege from u the condition on line 28 enables the fork to be sent from q to u. All of the above four cases contradict the fact that array $fork[\cdot]$ of u stops changing.

Consider the directed "waits-for" graph $W = (U, E_W)$ where vertices are nodes in U and (u, q) is in E_W if and only if $fork[q] = F$ holds at u in $E_1 \cap E_2$. Since each node in U is missing (waiting for) at least one fork, each vertex in W has at least one outgoing edge, and thus there is a cycle in W (basic fact from graph theory). This contradicts the total ordering of node ids since we just showed in the previous paragraph that $q > u$ holds for each edge $(u, q) \in E_W$. □

We now focus on showing that a node that is in a trying section but not inside the doorway eventually enters the doorway. The structure of the proof is analogous

to Lemma 2.3.10 and Lemma 2.3.11.

Lemma 2.3.12. *Suppose that there exists a failure-insulated node $p \in \Pi$. Also, suppose that there exists a suffix E_1 of E in which p does not set inside to T even if p is in a trying section and inside $= F$ holds at p. Then, there exists a suffix of E in which for all nodes $q \in N_p$, $ack[q]$ at p stops changing.*

Proof. First note that, by Lemma 2.3.8, *inside* $= F$ holds when p entered the trying section and the only way to set *inside* to T is by calling EnterDoorway() (which requires that all ack messages from p's neighbors to be gathered). In E_1, if node p ever receives an ack message from q, then $ack[q]$ remains as T until EnterDoorway() is called which proves the lemma. □

Lemma 2.3.13. *Consider each failure-insulated node $p \in \Pi$. If (1) Finite Eating holds for $E|X$, (2) p is in a trying section, and (3) inside $= F$ holds at p, then p eventually sets inside to T.*

Proof. Suppose, in contradiction, that there exists a non-empty set $U \subseteq \Pi$ of failure-insulated nodes such that for all nodes $u \in U$, u is in its trying section and *inside* $= F$ holds at u but u never sets *inside* to T. Then, by Lemma 2.3.12, there exists a suffix E_1 of E in which for all nodes in U, array $ack[\cdot]$ stops changing.

Since $E|D$ satisfies the $\Diamond P^1$ specification, there exists a suffix E_2 of E in which for all nodes $r \in \Pi$, every occurrence of event $response_r(S)$ returns $S \neq \emptyset$ if and only if r has a faulty neighbor. We only consider suffix $E_1 \cap E_2$ of E for our proof.

For each $u \in U$, there exists a node $q \in N_u$ in its trying section such that $deferred[u] = T$ holds at q; because otherwise all neighbors of u must have executed line 19 or lines 71 to 72 which results in u receiving ack messages from all of its neighbors. We show that q is in U and $last(E|X, q)$ occurs before $last(E|X, u)$.

Note that both $last(E|X,q)$ and $last(E|X,u)$ are *try* events by assumption. We first show that $q \in U$. Suppose,in contradiction, that $q \notin U$. If q has a faulty neighbor, then eventually q calls SatisfyRequests() which enables q to send an ack message to u, a contradiction. If q is failure-insulated and if ever q sets *inside* to T, enters a critical section, or enters an exiting section, then by Lemma 2.3.11 and the Well-formedness property (Lemma 2.3.3), q eventually calls SatisfyRequests() which again enables an ack message to be sent from q to u, a contradiction.

Now we show that $last(E|X,q)$ occurs before $last(E|X,u)$. Since $q \in U$, the only case that q could have set $deferred[u]$ to T for the last time is when $replied[u] = T$ (see the condition in line 16). This implies that when q sets $replied[u]$ to T for the last time, node q is in the trying section corresponding to $last(E|X,q)$. Note that an ack message $\langle ack \rangle$ is sent from q to u when q sets $replied[u]$ to T for the last time. By Lemma 2.3.9, q must have set $deferred[u]$ to T for the last time by receiving a ping message from u that was generated after u received message $\langle ack \rangle$. Hence, since a ping message is transmitted only when a *try* event occurs, $last(E|X,q)$ occurs before $last(E|X,u)$.

Consider the directed "waits-for" graph $W = (U, E_W)$ where vertices are nodes in U and (u,q) is in E_W if and only if $deferred[u] = T$ holds at q in $E_1 \cap E_2$. Since each node in U is missing (waiting for) at least one ack, each vertex in W has at least one outgoing edge, and thus there is a cycle in W (basic fact from graph theory). This contradicts the ordering of (try) events in E since we just showed in the previous paragraph that $last(E|X,q)$ occurs before $last(E|X,u)$ for each $(u,q) \in E_W$. □

Following lemma proves the $FL1$-progress property.

Lemma 2.3.14. $E|X$ *satisfies* FL1-progress.

Proof. Lemma 2.3.13 and Lemma 2.3.11 directly provide the proof. □

2.3.3.3 FL1-BW

The proof of Lemma 2.3.15 is similar to the proof of Theorem 3 in [68].

Lemma 2.3.15. *$E|X$ satisfies* FL1-*BW*.

Proof. Since $E|D$ satisfies the $\Diamond P^1$ specification, there exists a suffix E_1 of E in which for all nodes $r \in \Pi$, every occurrence of event $response_r(S)$ returns $S \neq \emptyset$ if and only if r has a faulty neighbor. Also, there exists a suffix E_2 of E_1 in which for all nodes $r \in \Pi$, event $response_r(S)$ occurs at least once. Let H be the set of nodes that are in a trying section at the beginning of E_2. Then, by $FL1$-progress (Lemma 2.3.14), there exists a suffix E_3 of E_2 in which all correct nodes in H enter a critical section. Hence, in E_3, it holds that for all correct processes $s \in \Pi$, s enters a trying section. We only consider E_3 in our proof.

Consider any two neighboring nodes $p \in \Pi$ and $q \in N_p$ where p is failure-insulated. First note that $skeptical = F$ holds at p in E_3.

By the code, while p is continuously in a trying section, node p sends at most one ack message to q; this is because when p receives a ping message from q for the first time during the trying section, p may immediately send an ack message to q (line 19) and set $replied[q]$ to T (line 21), and when p receives a ping message for the second time during the trying section, p may set $deferred[q]$ to T (line 17, since $replied[q] = T$ at p) instead of directly sending an ack message.

From node q's perspective, it may receive *two* ack messages from p while p is in the trying section. This may happen when p sent an ack message just before p enters the trying section; due to the assumption of unbounded message delay, this ack message is in transit when p enters a trying section. As a result, since receiving an ack message from p is a requirement for setting $inside$ to T at q (by calling EnterDoorway()) and $inside = T$ is a requirement for q to enter its critical section,

q may enter a critical section at most *two* times while p is in a trying section. This implies that, after the occurrence of event try_p, there can be at most two occurrences of event $crit_q$ before event $crit_p$ happens. □

Finally, we state the main theorem of this section.

Theorem 2.3.16. *The algorithm in Figures 2.1, 2.2, and 2.3 is an algorithm for the BW-□SX-FL1 problem.*

Proof. Let E be any execution of the algorithm in Figures 2.1, 2.2, and 2.3 such that $E|N$ satisfies the message-passing specification, either $E|D$ is not user-correct for D or $E|D$ satisfies the $\Diamond P^1$ specification, and $E|X$ is user-correct for X. Then, $E|X$ satisfies the BW-$\Box SX$-$FL1$ specification by Lemmas 2.3.3, 2.3.1, 2.3.7, 2.3.14, and 2.3.15. □

2.4 Extracting $\Diamond P^1$ from a Solution to BW-$\Box SX$-$FL1$

In this section, we present an algorithm that implements the $\Diamond P^1$ failure detector using multiple instances of BW-$\Box SX$-$FL1$. For each ordered pair (p, q) of nodes in Π, the algorithm uses instance $I(p, q)$ of problem BW-$\Box SX$-$FL1$ on the following graph $G^{I(p,q)}$ (see Figure 2.4):

- The vertex set $\Theta^{I(p,q)}$ of $G^{I(p,q)}$ consists of three threads on p and four threads on q.
- Two threads on p are called *hybrid* threads, denoted h_0 and h_1, and the remaining thread on p is called the *witness* thread, denoted w.
- The four threads on q that are in $\Theta^{I(p,q)} \backslash \{w, h_0, h_1\}$ are called *subject* threads, denoted $s_{i,j}$ for all $i, j \in \{0, 1\}$.
- The edge set of $G^{I(p,q)}$ is $\{\{h_0, h_1\}, \{h_i, w\}, \{s_{i,0}, s_{i,1}\}, \{h_i, s_{i.j}\} : i, j \in \{0, 1\}\}$.

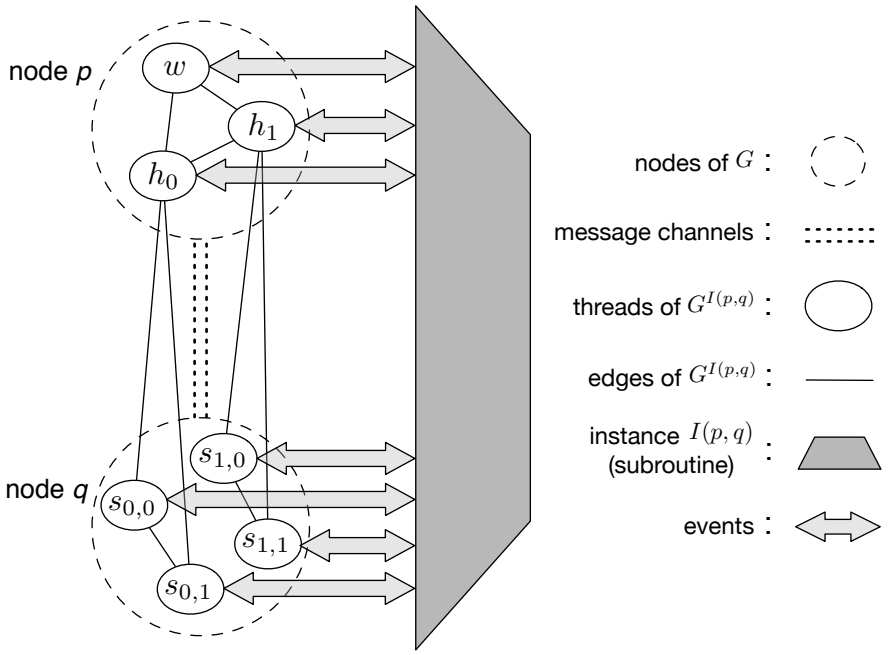


Figure 2.4: Instance $I(p,q)$ of BW-$\Box SX$-$FL1$ and graph $G^{I(p,q)}$.

For each instance, the witness thread (Figure 2.5) monitors hybrid threads (Figure 2.6) and subject threads (Figure 2.7) are monitored by hybrid threads.

The number of instances used in our construction of $\Diamond P^1$ on the communication topology graph G is $2|E|$ where E is the set of undirected edges of G.

When necessary for disambiguation, we will use $I(p,q)$ as a superscript (e.g., $h_1^{I(p,q)}, w^{I(p,q)}, rem_w^{I(p,q)}$). In the algorithm, the variables at the two hybrid threads on node p and the four subject threads on node q are disambiguated by subscripts. Variables are shared[9] between the two hybrid threads h_0 and h_1. Also, for all $i \in \{0,1\}$, the two subject threads $s_{i,0}$ and $s_{i,1}$ share variables.

For convenience, we assume the following for the algorithm:fl1dining: For each instance $I(p,q)$, there exists a FIFO channel between the witness thread and each

[9]We assume atomic (linearizable) shared variables.

Witness Thread w at node p:

(Variables and Initialization):

1: $counter \leftarrow 0$;
2: $val \leftarrow F$;
3: $bound \leftarrow 1$;
4: generate try_w;

5: $\langle$**When** rem_w occurs$\rangle$
6: generate try_w;

7: $\langle$**When** $crit_w$ occurs$\rangle$
8: $counter \leftarrow counter + 1$;
9: **if** $counter > bound$ **then**
10: $\quad bound \leftarrow counter$;
11: $\quad W[p][q] \leftarrow T$; // shared with d on node p
12: **else**
13: $\quad W[p][q] \leftarrow F$; // shared with d on node p
14: generate $exit_w$;

15: $\langle$**When** $recv_w(\langle ping \rangle, h_i)$ occurs$\rangle$ // h_i on node p
16: $counter \leftarrow 0$;
17: generate $send_w(\langle ack \rangle, h_i)$; // h_i on node p

Figure 2.5: Extracting $\Diamond P^1$ from BW-$\Box SX$-$FL1$; code for witness thread of instance $I(p, q)$ where $p \in \Pi$ and $q \in N_p$.

Hybrid Thread $h_{i\in\{0,1\}}$ at node p:

(Variables and Initialization):

1: $counter_i \leftarrow 0$;
2: $val_i \leftarrow F$;
3: $bound_i \leftarrow 1$;
4: $critical_i \leftarrow F$;
5: $ack_i \leftarrow F$; // shared with h_{1-i} on node p
6: generate try_{h_i};
7: generate $send_{h_i}(\langle ping\rangle, w)$; // w on node p

8: $\langle$**When** rem_{h_i} occurs$\rangle$
9: generate try_{h_i};
10: generate $send_{h_i}(\langle ping\rangle, w)$; // w on node p

11: $\langle$**When** $crit_{h_i}$ occurs$\rangle$
12: $critical_i \leftarrow T$;
13: $counter_i \leftarrow counter_i + 1$;
14: **if** $counter_i > bound_i$ **then**
15: $\quad bound_i \leftarrow counter_i$;
16: $\quad H[p][q][i] \leftarrow T$; // shared with d on node p
17: **else**
18: $\quad H[p][q][i] \leftarrow F$; // shared with d on node p

19: $\langle$**When** $recv_{h_i}(\langle ack\rangle, w)$ occurs$\rangle$ // w on node p
20: $ack_i \leftarrow T$;

21: $\langle$**When** $recv_{h_i}(\langle ping\rangle, s_{i,j})$ occurs$\rangle$ // $s_{i,j}$ on node q
22: $counter_i \leftarrow 0$;
23: generate $send_{h_i}(\langle ack\rangle, s_{i,j})$; // $s_{i,j}$ on node q

24: $\langle$**When** $critical_i \wedge ack_i \wedge ack_{1-i}\rangle$ // internal event
25: $ack_i \leftarrow F$;
26: $critical_i \leftarrow F$;
27: generate $exit_{h_i}$;

Figure 2.6: Extracting $\Diamond P^1$ from BW-$\Box SX$-$FL1$; code for hybrid threads of instance $I(p, q)$ where $p \in \Pi$ and $q \in N_p$.

Subject Thread $s_{i,j}$ at node q where $i, j \in \{0, 1\}$:

(Variables and Initialization):

1: $critical_{i,j} \leftarrow F$;
2: $ack_{i,j} \leftarrow F$; // shared with $s_{i,1-j}$ on node q
3: generate $try_{s_{i,j}}$;
4: generate $send_{s_{i,j}}(\langle ping \rangle, h_i)$; // h_i on node p

5: $\langle$**When** $rem_{s_{i,j}}$ occurs$\rangle$
6: generate $try_{s_{i,j}}$;
7: generate $send_{s_{i,j}}(\langle ping \rangle, h_i)$; // h_i on node p

8: $\langle$**When** $crit_{s_{i,j}}$ occurs$\rangle$
9: $critical_{i,j} \leftarrow T$;

10: $\langle$**When** $recv_{s_{i,j}}(\langle ack \rangle, h_i)$ occurs$\rangle$ // h_i on node p
11: $ack_{i,j} \leftarrow T$;

12: $\langle$**When** $critical_{i,j} \wedge ack_{i,j} \wedge ack_{i,1-j}\rangle$ // internal event
13: $ack_{i,j} \leftarrow F$;
14: $critical_{i,j} \leftarrow F$;
15: generate $exit_{s_{i,j}}$;

Figure 2.7: Extracting $\Diamond P^1$ from $BW\text{-}\Box SX\text{-}FL1$; code for subject threads of instance $I(p, q)$ where $p \in \Pi$ and $q \in N_p$.

hybrid thread.

There is also a *decision* thread, denoted as d, for each node $p \in \Pi$ (see Figure 2.8). It is important to note that there exists exactly one decision thread per node, *not* per instance. The decision thread on node p determines the parameter S (the suspicion set) of the output event $response_p(S)$. The decision thread at node p shares (atomic) variables with the witness and hybrid threads of instance $I(p, q)$ for all neighbors q of p.

Decision Thread d at node p:

(Variables and Initialization):

1: N_p; // neighbor set of p
2: $S \leftarrow N_p$;
3: $\forall q \in N_p : W[p][q] \leftarrow T$; // $W[p][q]$ is shared with w of $I(p, q)$
4: $\forall q \in N_p, \forall i \in \{0, 1\} : H[p][q][i] \leftarrow T$; // $H[p][q][i]$ is shared with h_i of $I(p, q)$

5: $\langle$**When** $query_p$ occurs$\rangle$
6: $S \leftarrow \{q : q \in N_p, W[p][q] = T\} \cup$
$\{r : r \in N_p, H[p][r][i] = T, i \in \{0, 1\}\}$;
7: generate $response_p(S)$;

Figure 2.8: Extracting $\Diamond P^1$ from BW-$\Box SX$-$FL1$; code for decision thread at node p.

For any given instance $I(p, q)$, hybrid threads h_0 and h_1 on node p monitor whether node q crashed. This monitoring activity of each hybrid thread is only possible if at least one of the hybrid threads does not starve. In the case when both h_0 and h_1 starve, the monitoring activity of the witness thread w detects the starvation of both hybrid threads. Note that, if node p is correct and the hybrid threads on p starve, then the $FL1$-progress property tells us that p has a faulty neighbor.

2.4.1 Algorithm Overview

2.4.1.1 Hybrid Thread $h_{i\in\{0,1\}}$ and Subject Threads $s_{i,0}$ and $s_{i,1}$ of $I(p, q)$

Subject thread $s_{i,j}$ sends a ping message to h_i while it is in its trying section. Note that after $s_{i,j}$ sends a ping message, it cannot send another ping message unless it enters its exit section. Also, $s_{i,j}$ cannot enter its exit section unless $s_{i,j}$ enters its critical section and both $s_{i,j}$ and $s_{i,1-j}$ receive an ack message from h_i; variable

$critical_{i,j}$ is used to keep track of whether $s_{i,j}$ is in its critical section and variable $ack_{i,j}$ is used to check whether $s_{i,j}$ received an ack message from hybrid thread h_i as a response to a ping message sent from $s_{i,j}$ to h_i. Moreover, just before entering its exiting section, $s_{i,j}$ sets $ack_{i,j}$ to F which prevents $s_{i,1-j}$ from entering its exit section until $s_{i,j}$ receives an ack message from h_i. This protocol guarantees that (1) at least one of $s_{i,0}$ or $s_{i,1}$ is in its trying or critical section when h_i receives a ping message and (2) at least one of $s_{i,0}$ or $s_{i,1}$ stays in its trying or critical section until h_i receives the next ping message.

Now we explain how witness thread h_i on node p detects that q crashed. First note that when q crashes, then all subject threads running on q crash as well. Each time h_i enters its critical section, it checks whether $counter_i$ is growing without bound by comparing $counter_i$ with variable $bound_i$; variable $bound_i$ keeps track of the maximum value of $counter_i$ since the beginning of the execution of the algorithm and it never decreases. Note that variable $counter_i$ is incremented when h_i enters a critical section. Also, note that, when hybrid thread h_i receives a ping message from a subject thread (either $s_{i,j}$ or $s_{i,1-j}$), it resets the $counter_i$ variable. Suppose nodes p and q are both correct. In this case, subject threads on q do not starve due to $FL1$-progress. Since there exists a subject thread that is continuously in its trying or critical section between any two consecutive ping message receptions at h_i, the $FL1$-BW property guarantees that $counter_i$ will be bounded. As a result, element $H[p][q][i]$ will be set to F from some time on. Now, suppose q is faulty. In this case, *if* h_i *does not starve*[10], then variable $counter_i$ grows without bound since subject threads will eventually stop sending ping messages to h_i. Consequently, element $H[p][q][i]$ will be set to T from some time on.

[10]The BW-$\Box SX$-$FL1$ specification only guarantees nodes that are at least two hops away from a failure to make progress; neighbors of a faulty thread may starve.

2.4.1.2 Witness Thread w and Hybrid Threads h_0 and h_1 of $I(p, q)$

The interaction between the witness thread and the hybrid threads is similar to the interaction between a hybrid thread and its subject threads except that, this time, witness thread w monitors whether *both* h_0 and h_1 starve or not; the monitoring is based on the fact that starving hybrid threads eventually stop sending ping messages to w. By the $FL1$-progress property, the existence of a starving hybrid thread on node p indicates that node q is faulty.

2.4.1.3 Decision Thread d at node p

The decision thread d at node p simply identifies each neighbor q of p where either $W[p][q] = T$ or $H[p][q][i] = T$ holds for all $i \in \{0, 1\}$, and passes the set of those identified neighbors of p, denoted S, to event $response_p$ as a parameter.

2.4.2 Proof Outline

In this section, we provide an outline of the proof of the algorithm in Figures 2.5, 2.6, 2.7, and 2.8.

The code of Figure 2.8 shows that a response event immediately follows a query event (lines 5–7). Thus, the Well-formedness (Lemma 2.4.1) and Liveness (Lemma 2.4.2) properties hold. To assist in proving both the Completeness property and the Accuracy property, we first show that for all correct neighboring nodes p and q, (1) our algorithm interacts with the $I(p, q)$ subroutine properly (Lemma 2.4.3) and (2) no thread in instance $I(p, q)$ stays in its corresponding critical sections forever (Lemma 2.4.4). For all correct neighboring nodes p and q, part (1) implies that instance $I(p, q)$ satisfies the BW-$\Box SX$-$FL1$ specification and part (2) implies that each thread in $I(p, q)$ that is in its trying section eventually enters its critical section.

The Completeness property is shown in the following way: Suppose a correct

node p has a faulty neighbor q and consider instance $I(p, q)$. The hybrid threads will eventually stop receiving ping messages from subject threads. If at least one hybrid thread does not starve, then the non-starving hybrid thread can detect the failure using the $FL1$-BW property (Case 1 of Lemma 2.4.5). However, there is no progress guarantee for the hybrid threads since their neighbor crashed. In the case when both hybrid threads starve, the witness thread can detect that both of the hybrid threads are starving using the $FL1$-BW property (Case 2 of Lemma 2.4.5). This is because each starving hybrid thread stops sending ping messages to the witness thread and it is guaranteed that the witness thread will progress by the the $FL1$-*progress* property (the witness thread is two hops away from subject threads in the conflict graph). Note that, by the $FL1$-*progress* property, the starvation of any hybrid thread indicates that q is faulty.

The Accuracy property is shown using the following fact: If two neighboring nodes p and q are both correct, then, at least one hybrid thread (resp. subject thread) of instance $I(p, q)$ is continuously in a trying section or a critical section during the time in between any two consecutive ping message receptions at the witness thread (resp. hybrid thread) of $I(p, q)$ (Lemmas 2.4.7 and 2.4.8). This allows the witness thread and each hybrid thread to obtain a bound on the number of times it enters the critical section while at least one of its neighboring thread is continuously trying to enter the critical section (Lemma 2.4.9). Note that such bounds exist due to the $FL1$-BW property.

2.4.3 Proof of Correctness

Let E be any execution of the algorithm in Figures 2.5, 2.6, 2.7, and 2.8 such that

- $E|N$ satisfies the message-passing specification,

- for each h, $1 \leq h \leq H$, either $E|X^h$ is not user-correct for X^h or $E|X^h$ satisfies the $BW\text{-}\Box SX\text{-}FL1$ specification, and
- $E|D$ is user-correct for D.

2.4.3.1 Well-formedness and Liveness

Lemma 2.4.1. *$E|D$ satisfies* Well-formedness.

Proof. Since user-correctness is assumed for $E|D$, it is sufficient to show that for all nodes $p \in \Pi$, within $(E|D)|p$, event $query_p$ (if any) is the only event in $D \backslash \{crash_p\}$ that immediately follows event $response_p(\cdot)$. By the code of Figure 2.8, a $response_p(\cdot)$ event occurs only after the occurrence of a $query_p$ event (lines 5–7). Hence, within $(E|X)|p$, event $response_p(\cdot)$ cannot immediately follow a $response_p(\cdot)$ event which proves the lemma. □

Lemma 2.4.2. *$E|D$ satisfies* Liveness.

Proof. It is direct from the code of Figure 2.8, that for all correct nodes $p \in \Pi$, whenever a $query_p$ event occurs, there is an immediate occurrence of $response_p(\cdot)$ (lines 5–7). □

2.4.3.2 Completeness

Before we prove the Completeness property, we show that for all nodes $p \in \Pi$ and $q \in N_p$, $E|X^{I(p,q)}$ is user-correct.

Lemma 2.4.3. *Consider each pair of neighboring nodes $p \in \Pi$ and $q \in N_p$. Then, $E|X^{I(p,q)}$ is user-correct.*

Proof. The proof requirement is to show that the shortest prefix of $E|X^{I(p,q)}$ to violate Well-formedness (if any) ends in either rem_x or $crit_x$ for some $x \in \Theta^{I(p,q)}$. Consider witness thread w at node p. Directly from the code in Figure 2.5, a try_w

(resp. $exit_w$) event only occurs immediately after the occurrence of a rem_w (resp. $crit_w$) event.

Consider hybrid thread $h_{i\in\{0,1\}}$ at node p. Again, directly from the code, a try_{h_i} event only occurs immediately after the occurrence of a rem_{h_i} event which implies that a try_{h_i} event does not immediately occur after a $crit_{h_i}$ event or an $exit_{h_i}$ event, and an $exit_{h_i}$ event does not immediately occur after a rem_{h_i} event. The proof requirement is to show that $exit_{h_i}$ does not immediately occur after an $exit_{h_i}$ event or a try_{h_i} event. Suppose, in contradiction, that an $exit_{h_i}$ event occurs immediately after an $exit_{h_i}$. Then, when the former $exit_{h_i}$ event occurs, variable $critical_i$ is set to F. For the latter $exit_{h_i}$ event to occur, $critical_i$ must be T. The only case that the hybrid thread can set $critical_i$ to T is by an occurrence of a $crit_{h_i}$ event. This implies that there must have been an occurrence of a $crit_{h_i}$ event in between the two $exit_{h_i}$ events, a contradiction. Now, suppose, in contradiction, that an $exit_{h_i}$ event occurs immediately after a try_{h_i} event. For an $exit_{h_i}$ event to occur, $critical_i = T$ must hold which shows that there exists an occurrence of a $crit_{h_i}$ event before the $exit_{h_i}$ event happens. Let $crit'_{h_i}$ be the latest $crit_{h_i}$ event that happened before the occurrence of an $exit_{h_i}$ event. Then, in between $crit'_{h_i}$ and the $exit_{h_i}$ event there must exist a try_{h_i} event which immediately follows a rem_{h_i} event. If a rem_{h_i} event did not happen immediately after $crit'_{h_i}$, the only event that can occur immediately after $crit'_{h_i}$ is a try_{h_i} event which contradicts the fact that a try_{h_i} event only occurs immediately after a rem_{h_i} event.

The proof for the subject threads is similar to the proof for hybrid threads. □

Now, we show that for all nodes $p \in \Pi$ and $q \in N_p$, if p and q are both correct, then Finite Eating holds for $E|X^{I(p,q)}$.

Lemma 2.4.4. *Consider each pair of neighboring nodes $p \in \Pi$ and $q \in N_p$ where p*

is correct. Then, Finite Eating holds for $E|X^{I(p,q)}$.

Proof. Consider witness thread w at node q. Directly from the code in Figure 2.5, there exists an immediate occurrence of an $exit_w$ event after a $crit_w$ event.

Now, without loss of generality, consider hybrid thread h_0 at node p. Note that as long as user-correctness (preserving Well-formedness) holds, a correct algorithm for solving $BW\text{-}\Box SX\text{-}FL1$ cannot violate Exclusion or Finite Exiting. Suppose, in contradiction, hybrid thread h_0 is in a critical section forever. Then, there exists a last $crit_{h_0}$ event that occurs. Let E_1 be the suffix of E that begins with this event $crit_{h_0}$. Then, in E_1, variable $critical_0$ of thread h_0 is always T since the only case that variable $critical_0$ can be set to T is when h_0 enters its critical section and the only case that $critical_0$ can be reset to F is when h_0 enters an exiting section (lines 12 and 26 of Figure 2.6). Since $E|X^{I(p,q)}$ is user-correct by Lemma 2.4.3, there exists a latest try_{h_0} event that occurs before the event $crit_{h_0}$. By the code of Figure 2.6, immediately after h_0 enters a trying section, a ping message is sent to the witness thread w. By reliable message delivery, this ping message will arrive at w and w will immediately send back an ack message to h_0 (line 17 of Figure 2.5). Again by reliable message delivery, h_0 will receive the ack message from w. Let E_2 be the suffix of E that begins with the $recv$ event which corresponds to this ack message reception. Then, in E_2, variable ack_0 is forever T since the only case that variable ack_0 can be set to T is when h_0 receives an ack message and the only case that ack_0 can be reset to F is when h_0 enters an exiting section (lines 20 and 25 of Figure 2.6).

Considering hybrid thread h_1, note that, by the previous paragraph, h_1 cannot enter its critical section infinitely often because Exclusion will be violated (since h_0 is in its critical section forever). Since p is correct and $E|X^{I(p,q)}$ is user-correct (by Lemma 2.4.3), h_1 must be trapped in either a remainder, trying, critical, or

exiting section forever. Thread h_1 cannot be in its critical section forever, as it violates Exclusion (since h_0 is in a critical section forever). Thread h_1 cannot be in its exiting section forever, since Finite Exiting holds. Thread h_1 cannot be in its remainder section forever since, by lines 8–10 of Figure 2.6, a try_{h_1} event immediately follows a rem_{h_1} event. Thus, h_1 must be in its trying section forever.

By the code, when the last try_{h_1} event occurs at h_1, thread h_1 sends a ping message to w and w will send back an ack message $\langle ack \rangle$ (by reliable message delivery and line 17 of Figure 2.5). Let E_3 be the suffix of E that begins with the $recv_{h_1}$ event that corresponds to the reception of $\langle ack \rangle$. Then, in E_3, variable ack_1 is forever T since ack_1 can only be reset to F when an $exit_{h_1}$ event occurs (lines 25–27 of Figure 2.6). Hence, in $E_1 \cap E_2 \cap E_3$, the precondition on line 24 of Figure 2.6 for h_0 to exit a critical section (and enter an exiting section) is always T. This contradicts the assumption that h_0 is in its critical section forever.

The proof for the subject threads is similar to the proof for hybrid threads. □

The following lemma shows that if a correct node p has faulty neighbor q, then from some time on, every occurrence of event $response_p(S)$ has $q \in S$.

Lemma 2.4.5. *Consider each correct node $p \in \Pi$. If neighbor q of p is faulty, then there exists a suffix of E such that every occurrence of event $response_p(S)$ has $q \in S$.*

Proof. By Lemma 2.4.3, $E|X^{I(p,q)}$ is user-correct. Thus, we may assume that $E|X^{I(p,q)}$ satisfies the BW-$\Box SX$-$FL1$ specification for $G^{I(p,q)}$.Also, since Finite Eating holds for $E|X^{I(p,q)}$ by Lemma 2.4.4, the $FL1$-progress property of the BW-$\Box SX$-$FL1$ specification guarantees that, for all threads τ in $G^{I(p,q)}$ that are two hops away from a crashed thread in $G^{I(p,q)}$, if τ is in its trying section, then τ will eventually enter its critical section.

For the proof, we only consider instance $I(p, q)$. If q is faulty, then there exists a suffix E_1 of E such that each subject thread on q stops sending ping messages to hybrid threads on p. Consequently, there exists a suffix E_2 of E such that each hybrid thread on p stops receiving ping messages from subject threads on q. Depending on the behavior of the hybrid threads, we divide into two cases.

–(*Case* 1) At least one hybrid thread enters its critical section infinitely often: Without loss of generality, let h_0 be the hybrid thread that enters it critical section infinitely often. Note that the only way to decrease the value of $counter_0$ at h_0 is by receiving a ping message from a subject thread which implies that, in E_2, $counter_0$ does not decrease. In E_2, whenever $crit_{h_0}$ occurs, $counter_0$ at h_0 is incremented by 1. Thus, eventually, $counter_0$ will exceed $bound_0$ at h_0 since $counter_0$ never decreases in E_2.

Let E_3 be a suffix of E_2 that begins with a $crit_{h_0}$ event where $counter_0$ exceeds $bound_0$ for the first time after the beginning of E_2. Since variable $bound_0$ is set to the current value of $counter_0$ whenever $counter_0$ exceeds $bound_0$, variable $counter_0$ will exceed $bound_0$ whenever $crit_{h_0}$ occurs in E_3. As a result, $H[p][q][0]$ will be set to T whenever $crit_{h_0}$ happens in E_3. Hence, in E_3, set S of the decision thread d on node p will include q permanently (line 6 of Figure 2.8). Consequently, every occurrence of event $response_p(S)$ in E_3 will have $q \in S$.

–(*Case* 2) Both hybrid threads of $I(p, q)$ starve: In this case, there exists a suffix E_4 of E such that both hybrid threads h_0 and h_1 (1) remain in a certain non-critical section forever, and (2) do not send ping messages to the witness thread w. From w's perspective, this means that there exists a suffix E_5 of E such that it does not receive any ping messages from either h_0 or h_1. As a result, w does not reset *counter* to 0 in E_5 (line 16 of Figure 2.5). Note that the only way to decrease the value of *counter* at w is by receiving a ping message from a subject thread which implies

that, in E_5, the value of *counter* at w does not decrease.

Since p is correct, by $FL1$-progress and Finite Exiting, and by the fact that event try_w (resp. $exit_w$) occurs immediately after the occurrence of event rem_w (resp. $crit_w$) (lines 6 and 14 of Figure 2.5), event $crit_w$ occurs infinitely often at w. In E_5, whenever $crit_w$ occurs, *counter* at w is incremented by 1. Thus, eventually, *counter* exceeds *bound* since *counter* never decreases in E_5. The rest of the proof is similar to (Case 1) except that, in this case, $W[p][q]$ is set to T from some time on. □

The Completeness property directly follows from Lemma 2.4.5.

Lemma 2.4.6. *$E|D$ satisfies* Completeness.

2.4.3.3 Accuracy

As in [69], we define what is called a *renewal interval*. For all nodes $p \in \Pi$ and $q \in N_p$, let $W_n^{I(p,q)}$ be the n-th occurrence of a *ping* message reception at witness thread $w^{I(p,q)}$. Then, for $n \geq 1$, we denote $E[W_n^{I(p,q)}, W_{n+1}^{I(p,q)}]$ as an execution segment of E that begins with $W_n^{I(p,q)}$ and ends with $W_{n+1}^{I(p,q)}$ and we call this the n-th renewal interval of $w^{I(p,q)}$ in E for instance $I(p,q)$.

For a particular thread τ of instance $I(p,q)$, we call an execution segment E_τ of E a *trying-critical* section of τ if

- E_τ begins with a try_τ event,
- E_τ ends with an $exit_\tau$ event,
- exactly one $crit_\tau$ occurs in E_τ, and
- no event in $X^{I(p,q)}$ of τ, except the above three events, occurs in E_τ.

Next we show that each renewal interval of a witness thread $w^{I(p,q)}$ is encompassed by a trying-critical section of a subject thread $h_{i\in\{0,1\}}^{I(p,q)}$ if nodes p and q are both correct

and if p has no faulty neighbors. The proof of Lemma 2.4.7 is similar to the proof of Lemma 1 in [69].

Lemma 2.4.7. *Consider each pair of correct neighboring nodes $p \in \Pi$ and $q \in N_p$. Then, for all $n \geq 1$, (1) every renewal interval $E[W_n^{I(p,q)}, W_{n+1}^{I(p,q)}]$ is encompassed by a trying-critical section of a hybrid thread $h_{i\in\{0,1\}}^{I(p,q)}$ which does not send the ping message that enables the occurrence of $W_{n+1}^{I(p,q)}$, and (2) for each renewal interval $E[W_n^{I(p,q)}, W_{n+1}^{I(p,q)}]$, both hybrid threads $h_0^{I(p,q)}$ and $h_1^{I(p,q)}$ are in a trying-critical section when $W_{n+1}^{I(p,q)}$ occurs.*

Proof. By Lemma 2.4.3, $E|X^{I(p,q)}$ is user-correct. Hence, we may assume that $E|X^{I(p,q)}$ satisfies the BW-$\Box SX$-$FL1$ specification for $G^{I(p,q)}$. Also, since Finite Eating holds for $E|X^{I(p,q)}$ by Lemma 2.4.4, the $FL1$-progress property of the BW-$\Box SX$-$FL1$ specification guarantees that, for all threads τ in $G^{I(p,q)}$ that are two hops away from a crashed thread in $G^{I(p,q)}$, if τ is in its trying section, then τ will eventually enter its critical section. We prove by induction on n.

–(**Base case**; $n = 1$) Without loss of generality, let h_0 be the hybrid thread that sent the ping message which enabled the occurrence of $W_1^{I(p,q)}$; note that a ping message is sent immediately after entering a trying section. Afterwards, h_0 does not leave its critical section until h_0 receives an ack message from w. By the Well-formedness property, this implies that h_0 is in a trying-critical section when $W_1^{I(p,q)}$ occurs .

For h_0 to leave the trying-critical section, h_1 must receive an ack message from w and set ack_1 to T. For h_1 to receive an ack message from w, h_1 must send a ping message to w since w sends an ack message only when it receives a ping message (lines 15–17 of Figure 2.5). This implies that the occurrence of $W_2^{I(p,q)}$ is a result of w receiving a ping message from h_1. Hence, $E[W_1^{I(p,q)}, W_2^{I(p,q)}]$ is encompassed by

the trying-critical section of h_0 and h_0 did not send the ping message that enabled the occurrence of $W_2^{I(p,q)}$. Also, since h_1 did not receive an ack message when $W_2^{I(p,q)}$ occurred, h_1 is also in a trying-critical section when $W_2^{I(p,q)}$ occurred.

–(**Inductive case**) Suppose the lemma holds for the n-th renewal interval where $n \geq 1$. We prove for the $n+1$-th renewal interval.

Without loss of generality, let h_0 be the subject thread that sent the ping message which enabled the occurrence of $W_n^{I(p,q)}$. Note that, by the inductive hypothesis, both h_0 and h_1 are in a trying-critical section when $W_n^{I(p,q)}$ occurred. We first show that the occurrence of $W_{n+1}^{I(p,q)}$ exists. Suppose, in contradiction, that the occurrence of $W_{n+1}^{I(p,q)}$ does not exist. Then, since the Well-formedness property of $E|X^{I(p,q)}$ holds, $h_{i\in\{0,1\}}$ must be trapped in either a remainder, trying, critical, or exiting section forever. Thread h_i cannot be in its remainder section forever because line 9 of Figure 2.6 directly shows that a try_{h_i} event immediately follows a rem_{h_i} event. Thread h_i cannot be in its critical, trying, or exiting section forever since Finite Eating, $FL1$-progress, and Finite Exiting holds. Hence, we have a contradiction.

Since we know that the occurrence of $W_{n+1}^{I(p,q)}$ exists, there exists a hybrid thread that enters an exiting section during $E[W_n^{I(p,q)}, W_{n+1}^{I(p,q)}]$. Without loss of generality, during $E[W_n^{I(p,q)}, W_{n+1}^{I(p,q)}]$, suppose h_0 enters an exiting section earlier than h_1 does. Also, let e' indicate the earliest $exit_{h_0}$ event during $E[W_n^{I(p,q)}, W_{n+1}^{I(p,q)}]$. Note that when e' occurs, variable ack_0 is set to F which prevents h_1 from entering an exiting section if it were in a critical section. Also, note that after ack_0 is set to F, it cannot be reset to T unless h_0 sends a ping message to w and receives the corresponding ack message from w. As a result, h_0 must have sent the ping message which enabled the occurrence of $W_{n+1}^{I(p,q)}$ and, in between the occurrences of e' and $W_{n+1}^{I(p,q)}$, hybrid thread h_1 cannot enter an exiting section. In addition, since e' is the first $exit$ event in $E[W_n^{I(p,q)}, W_{n+1}^{I(p,q)}]$, h_1 does not enter an exiting section in between the occurrence of

$W_n^{I(p,q)}$ and the occurrence of e'. Hence, h_1 remains in a trying-critical section during $E[W_n^{I(p,q)}, W_{n+1}^{I(p,q)}]$ and it does not send the ping message that enables the occurrence of $W_{n+1}^{I(p,q)}$. Also, since a ping message is sent when h_0 enters a trying section and since h_0 did not receive an ack message when $W_{n+1}^{I(p,q)}$ occurs, hybrid thread h_0 is also in a trying-critical section when $W_{n+1}^{I(p,q)}$ occurs. Lemma 2.4.7 holds. □

Now, for all nodes $p \in \Pi$ and $q \in N_p$, let $H_{i,n}^{I(p,q)}$ be the n-th occurrence of a *ping* message reception at hybrid thread $h_i^{I(p,q)}$ where $i \in \{0, 1\}$. For $n \geq 1$, we denote $E[H_{i,n}^{I(p,q)}, H_{i,n+1}^{I(p,q)}]$ as an execution segment of E that begins with $H_{i,n}^{I(p,q)}$ and ends with $H_{i,n+1}^{I(p,q)}$ and we call this the n-th renewal interval of $h_i^{I(p,q)}$ in E for instance $I(p, q)$.

Using similar arguments as in the proof of Lemma 2.4.7, we can show that the following lemma holds:

Lemma 2.4.8. *Consider each pair of correct neighboring nodes $p \in \Pi$ and $q \in N_p$. Then, for all $n \geq 1$, (1) every renewal interval $E[H_{i,n}^{I(p,q)}, H_{i,n+1}^{I(p,q)}]$ is encompassed by a trying-critical section of a subject thread $s_{i,j}^{I(p,q)}$ where $j \in \{0, 1\}$ which does not send the ping message that enables the occurrence of $H_{i,n+1}^{I(p,q)}$, and (2) for each renewal interval $E[H_{i,n}^{I(p,q)}, W_{i,n+1}^{I(p,q)}]$, both subject threads $h_{i,0}^{I(p,q)}$ and $h_{i,1}^{I(p,q)}$ are in a trying-critical section when $H_{i,n+1}^{I(p,q)}$ occurs.*

Now, we prove that if neighboring nodes p and q are both correct, then from some time on, every occurrence of event $response_p(S)$ satisfies $q \notin S$. The proof of Lemma 2.4.9 is similar to the proof of Theorem 2 in [69].

Lemma 2.4.9. *Consider each pair of correct neighboring nodes $p \in \Pi$ and $q \in N_p$. Then, there exists a suffix of E in which every occurrence of event $response_p(S)$ satisfies $q \notin S$.*

Proof. By Lemma 2.4.3, $E|X^{I(p,q)}$ is user-correct. Hence, we may assume that $E|X^{I(p,q)}$ satisfies the BW-$\Box SX$-$FL1$ specification for $G^{I(p,q)}$. Also, since Finite Eating holds for $E|X^{I(p,q)}$ by Lemma 2.4.4, the $FL1$-progress property of the BW-$\Box SX$-$FL1$ specification guarantees that, for all threads τ in $G^{I(p,q)}$ that are two hops away from a crashed thread in $G^{I(p,q)}$, if τ is in its trying section, then τ will eventually enter its critical section.

By the $FL1$-BW property, there exists an integer $k > 0$ and there exists a suffix E_1 of E such that every infix σ_1 of E_1 that (1) starts with $try^{I(p,q)}_{h_{i\in\{0,1\}}}$ and ends with $crit^{I(p,q)}_{h_{i\in\{0,1\}}}$ and (2) contains exactly one occurrence of $crit^{I(p,q)}_{h_{i\in\{0,1\}}}$, contains at most k occurrences of $crit^{I(p,q)}_{w}$.

Let e_1 and e_2 be the event that initiates the first renewal interval of $w^{I(p,q)}$ in E_1 and the first *try* event of $s^{I(p,q)}_{i\in\{0,1\}}$ in E_1, respectively (since $FL1$-progress, Finite Eating, and Finite Exiting hold, events e_1 and e_2 exist). Then, by Lemma 2.4.7, each renewal interval RI of $w^{I(p,q)}$ that begins after the occurrence of both e_1 and e_2 is encompassed by a trying-critical section of $s^{I(p,q)}_{i\in\{0,1\}}$ and, within RI, $w^{I(p,q)}$ can enter a critical section at most k times. This implies that, within RI, $w^{I(p,q)}$ may enter an exiting section at most $k+1$ times which in turn implies that the value of *counter* at $w^{I(p,q)}$ after the occurrence of both e_1 and e_2 is bounded by $k+1$.

Without loss of generality, suppose e_2 occurs after e_1. Then, before the occurrence of e_2, there exists a maximum value c_m of variable *counter* at $w^{I(p,q)}$. Thus, variable *counter* at $w^{I(p,q)}$ is bounded by $\max(c_m, k+1)$ in E and, due to lines 9 and 10 of Figure 2.5, variable *bound* at $w^{I(p,q)}$ is also bounded by $\max(c_m, k+1)$ in E. Since variable *bound* at $w^{I(p,q)}$ is bounded and never decreases and since we already know, from the proof of Lemma 2.4.5, that $crit^{I(p,q)}_{w}$ occurs infinitely often, there exists a suffix E_2 of E in which variable $W[p][q]$ on node p is set to F permanently.

Using similar arguments as above along with Lemma 2.4.8, we can show that

there exists a suffix of E in which, for all $i \in \{0, 1\}$, variable $W[p][q][i]$ on node p is set to F permanently (in this case, a hybrid thread h_i corresponds to the above witness thread and two subject threads $s_{i,0}$ and $s_{i,1}$ correspond to the above two hybrid threads).

Hence, we can conclude that there exists a suffix E' of E such that set S of the decision thread d on node p will *not* include q permanently (line 6 of Figure 2.8). Consequently, every occurrence of event $response_p(S)$ in E' satisfies $q \notin S$ which proves the lemma. □

The Accuracy property directly follows from Lemma 2.4.9.

Lemma 2.4.10. *$E|D$ satisfies* Accuracy*.*

Finally, we state the main theorem of this section.

Theorem 2.4.11. *The algorithm in Figures 2.5, 2.6, 2.7, and 2.8 is an algorithm for the $\Diamond P^1$ problem.*

Proof. Let E be any execution of the algorithm in Figures 2.5, 2.6, 2.7, and 2.8 such that

- $E|N$ satisfies the message-passing specification,
- for each h, $1 \leq h \leq H$, either $E|X^h$ is not user-correct for X^h or $E|X^h$ satisfies the $BW\text{-}\Box SX\text{-}FL1$ specification, and
- $E|D$ is user-correct for D.

Then, $E|D$ satisfies the $\Diamond P^1$ specification by Lemmas 2.4.1, 2.4.2, 2.4.6, and 2.4.10. □

2.5 Discussion

2.5.1 Failure Locality and Exclusion Guarantees

As already mentioned, $\Diamond P^1$ is known to be a weakest failure detector for wait-free *eventual* weak exclusion [62, 65]. The work in [65] discusses wait-free *perpetual* weak exclusion and shows that the trusting failure detector $\mathcal{T}$ (refer to [22, 65] for the specification) is necessary but not sufficient to solve wait-free perpetual weak exclusion. It is known that $\Diamond P$ is strictly weaker than $\mathcal{T}$ [22]. From these two results, we can notice that it is more expensive to achieve wait-free perpetual weak exclusion than wait-free eventual weak exclusion since the former requires a more powerful failure detector.

Our result shows that $\Diamond P^1$ is sufficient to solve failure-locality-1 perpetual strong exclusion. Since perpetual strong exclusion implies perpetual weak exclusion, $\Diamond P^1$ is sufficient to solve *failure-locality-1* perpetual weak exclusion. If we compare this to the result for *wait-free* perpetual weak exclusion, we can observe that wait-freedom (i.e. failure-locality-0) is more expensive to achieve than failure-locality-1 when perpetual weak exclusion is considered.

2.5.2 Bounded Waiting

The algorithm in Figures 2.1, 2.2, and 2.3 actually satisfies a property stronger than $FL1$-BW which we call $FL1$-$\Diamond k$-BW: there exists an integer $k > 0$ that holds for *all* executions such that if process p only has correct neighbors, then eventually, for any interval in which p is trying to enter its critical section, no neighbor of p enters its critical section more than k times. The algorithm satisfies $FL1$-$\Diamond k$-BW with $k = 2$. Note that $FL1$-$\Diamond k$-BW implies $FL1$-BW. Since $\Diamond P^1$ can be extracted from multiple instances of any solution to our dining problem that satisfies $FL1$-BW and since our dining algorithm uses $\Diamond P^1$ to solve $FL1$-$\Diamond k$-BW, $FL1$-BW and $FL1$-

$\Diamond k$-BW are mutually reducible (without preserving the underlying topology) in our setting.

2.5.3 Mapping from an Instance to Participating Processes

Consider process p on G and consider instance $I(p, q)$ and the conflict graph $G^{I(p,q)}$ for any $q \in N_p$. Note that the algorithm in Figures 2.5, 2.6, 2.7 and 2.8 assume that all threads of $I(p, q)$ know which processes are participating in $I(p, q)$. What if the threads of $I(p, q)$ do not know which processes are participating in $I(p, q)$? In other words, what if the witness thread and hybrid threads of $I(p, q)$ on process p do not know that the subject threads of $I(p, q)$ are located at process q and vice versa? In this case, it might not be possible for the witness and hybrid threads to identify that q has crashed. However, it is still possible for the threads of $I(p, q)$ on process p to identify that *some* neighbor of p has crashed. This observation implies that the construction in Section 2.4 can be used to extract the following failure detector, named as the "local anonymous eventually perfect failure detector" and denoted as $?\Diamond P^1$, even when the threads of a certain instance do not know the participating processes of that instance:

- From some time on, $?\Diamond P^1$ outputs *true* if there exists a crashed neighboring process.
- From some time on, $?\Diamond P^1$ outputs *false* if all neighboring processes are correct.

Note that $?\Diamond P^1$ is weaker than $\Diamond P^1$ ($?\Diamond P^1$ can be implemented with the use of $\Diamond P^1$) since it simply outputs a boolean (instead of process ids) regarding the existence of at least one crashed process within the neighborhood. $?\Diamond P^1$, which is a variant of the anonymous perfect failure detector introduced in [32], falls into the category of identity-free failure detectors [8]. The boolean-valued failure detectors in [8, 7, 23] and [32] provide system-wide information (e.g. am I the leader? [8], am I the only

correct process? [7, 23], is there a crashed process? [32]) whereas $?\Diamond P^1$ provides *local* information regarding the existence of a crashed neighbor.

Now the question is "can we use $?\Diamond P^1$ to solve $BW\text{-}\Box SX\text{-}FL1$?". It is easy to verify that the answer is *yes*: the concept of skepticism directly tells us that the required information from the failure detector is whether there exists a failure, not who failed. Thus, we can conclude that $?\Diamond P^1$ and $BW\text{-}\Box SX\text{-}FL1$ are mutually reducible (without preserving the underlying topology) when the mapping from an instance to the processes that participate in that instance is not given.

In Section 3, we use the $?\Diamond P^1$ failure detector to design a failure-locality-1 dining algorithm that tolerates unexpected corruptions to the system states.

3. STABILIZING DINING PHILOSOPHERS WITH FAILURE LOCALITY 1

In this section, similar to Section 2, we consider failure-locality-1 dining where we require that (1) eventually, no two neighbors in the conflict graph enter their corresponding critical sections simultaneously, and (2) each correct process that is trying to enter its critical section eventually does so if it is at least two hops away from any other crashed process in the conflict graph. However, in addition to crash failures, we take into account the presence of transient failures. Transient failures corresponds to unexpected corruptions to the system state; the system can be in an arbitrary system state after a transient failure occurs. Transient fault-tolerant algorithms are also known as *stabilizing* algorithms. In designing distributed algorithms, achieving transient and crash fault tolerance together is more difficult than achieving either one of them separately, as for instance, recovery from a transient failure might be disrupted by a later crash failure.

We consider a shared-memory system where processes communicate via read/write operations on shared *regular* registers. Regularity states that each read operation returns the value of some overlapping write operation or of the latest preceding write operation.

As mentioned in Section 2, Choy and Singh [14] showed that any asynchronous message-passing algorithm that solves dining must have failure locality at least 2. Although the failure-locality-2 lower bound in [14] is proved for asynchronous message-passing systems, it also applies to asynchronous shared-memory systems. This implies that failure-locality-1 dining cannot be solved in pure asynchrony. To circumvent this lower bound, we augment the shared-memory system with the local anonymous eventually perfect failure detector $?\Diamond P^1$ mentioned in Section 2.5, and

show that this failure detector is sufficient to solve the problem at hand.

We propose an algorithm for solving stabilizing failure-locality-1 dining in asynchronous shared-memory systems with regular registers. The algorithm is inspired by the Asynchronous Doorway (ADW) algorithm in [13]. Our algorithm utilizes stabilizing mutual exclusion subroutines which can be implemented using regular registers (e.g., Dijkstra's stabilizing token circulation algorithm using regular registers [25]).

3.1 Contributions

We present the problem specification for stabilizing failure-locality-1 dining; this specification the first to consider both failure locality 1 and stabilization. We present the first stabilizing failure-locality-1 dining algorithm in asynchronous shared-memory systems using failure detectors along with shared regular registers. The proposed algorithms are modular in the sense that they utilize stabilizing mutual exclusion subroutines.

3.2 Background and Related Work

After the early non-fault-tolerant dining algorithms [12, 24, 49] were introduced, there has been a significant body of work which considers fault-tolerant dining.

Stabilizing dining algorithms are presented in [4, 5, 10, 55, 59]. These algorithms all consider read/write atomicity and are not crash fault tolerant. We assume the use of regular registers, which are weaker than atomic registers. In addition, our dining algorithms are crash fault tolerant.

Dining algorithms that consider both crash fault tolerance and stabilization are presented in [57, 58, 66]. The dining algorithms in [57, 58] achieve failure locality 2. A wait-free (failure-locality-0) dining algorithm is presented in [66] which utilizes $\Diamond P$. We fill in the gap between wait-freedom and failure locality 2 by presenting a failure-locality-1 stabilizing dining algorithm.

In Section 3, we use the local anonymous eventually perfect failure detector $?\Diamond P^1$ and show that $?\Diamond P^1$ is sufficient to solve stabilizing failure-locality-1 dining. The $?\Diamond P^1$ failure detector returns a boolean value and has the following property: eventually, $?\Diamond P^1$ at process i returns true if and only if there exists a crashed neighboring process of i. The $?\Diamond P^1$ failure detector can be implemented using $\Diamond P$ in asynchronous systems. This means that the failure detector ($?\Diamond P^1$) that we are using to solve stabilizing failure-locality-1 dining is at most as powerful as the failure detector ($\Diamond P$) used in [66].

The correctness of our stabilizing failure-locality-1 dining algorithms relies on the information provided by the underlying failure detector. Since we consider systems prone to transient failures, the failure detector that we utilize must be stabilizing as well. Here, we list previous work on stabilizing failure detector implementations. A stabilizing version of $\Diamond P$ is implemented in [46, 47] considering a message-passing system in which at most one process can crash. Multiple crash failures are considered in [6, 35]. Both [6] and [35] assume a message-passing system and the existence of a bound on relative message delays in implementing a stabilizing version of the $\Diamond P^1$ failure detector (see Section 2 for the specification of $\Diamond P^1$); however, in [6], each process utilizes its local clock to send heartbeat messages while the implementation in [35] eliminates the use of local clocks.

There are several implementations of stabilizing failure detectors other than $\Diamond P$. Stabilizing implementations of the Ω failure detector[1] are presented in [21] and [26] considering the message-passing model and the shared-memory model, respectively. A stabilizing implementation of the $\Omega?$ failure detector, which is a variant of Ω that eventually detects whether or not there exists a leader, is presented in [28] considering

[1]The Ω failure detector, which outputs a single process ID, has the following property: eventually, every correct process outputs the process ID of some unique correct process forever.

the population-protocol model[2] [3].

3.3 System Model and Problem Specification

We consider a system that contains a set Π of n (dining) processes, where each process is a state machine. Each process has a unique incorruptible ID and is known to all the processes in the system. For convenience, we assume that the IDs form the set $\{0, \ldots, n-1\}$; we refer to a process and its ID interchangeably. There is an undirected graph G with vertex set Π, called the *(dining) conflict graph*. If $\{i, j\}$ is an edge of G, then we say that i and j are *neighbors*.

The *state* of a process i is modeled with a set of local variables, which we now discuss.

Each process i has a local variable $diningState_i$ through which it communicates with the user of the dining philosophers algorithm. The user sets $diningState_i$ to "hungry" to indicate that it needs exclusive access to the set of resources for i. Sometime later, the process should set $diningState_i$ to "eating", which is observed by the user. While $diningState_i$ is "eating", the user accesses its critical section. When the user is through eating, it sets $diningState_i$ to "exiting" to tell i that it can do some cleaning up, after which i should set $diningState_i$ to "thinking". This sequence of updates to $diningState_i$ can then repeat cyclically.

Process i has another local variable $?\Diamond P^1_i$ through which it communicates with the failure detector $?\Diamond P^1$ (the "local anonymous eventually perfect failure detector"). This variable is set to true or false at appropriate times by the failure detector module and is read (but never set) by process i. Recall from Section 2.5that the behavior of the failure detector is that after some time, $?\Diamond P^1_i$ is always false if i has no crashed

[2]In the population-protocol model, each process is modeled as a finite-state mobile sensor called an agent. Distributed computation is carried out by agents meeting each other; two agents can interact with each other only in the case when they meet. The interacting pattern is described by a directed graph called the interaction graph.

neighbors and is always true if i has at least one crashed neighbor.

The processes have access to a set of shared single-writer single-reader registers that satisfy the consistency condition of regularity, through which they can communicate. Reads and writes on such registers are not instantaneous. Each operation is invoked at some time and provides a response later. *Regularity* means that each read returns the value of some overlapping write or of the latest preceding write. If there is no preceding write, then any value can be returned. When a process invokes an operation on a shared register, it blocks until receiving the response.

Certain subsets of processes synchronize among themselves using mutual exclusion modules (i.e., subroutines). For any mutual exclusion module X, the participants in X are all neighbors of each other in the dining conflict graph. For each mutual exclusion module X in which it participates, (dining) process i has a local variable $X.mutex_i$. The mutual exclusion module X and process i communicate via $X.mutex_i$ in somewhat the opposite way that $diningState_i$ is used to communicate between i and the dining user (since i is the user of the mutual exclusion module). Process i, at an appropriate time, sets $X.mutex_i$ to "trying" when it needs access to the corresponding critical section. Subsequently, the mutual exclusion module should set $X.mutex_i$ to "critical". When i no longer needs the critical section for this mutual exclusion module, $X.mutex_i$ is set to "exiting", and at some later point the mutual exclusion module X should set the variable to "remainder". This sequence of updates to $X.mutex_i$ can then repeat cyclically. Note that such stabilizing mutual exclusion algorithms exist considering asynchronous shared-memory systems with regular registers (e.g. a variation of Dijkstra's stabilizing token circulation algorithm using regular registers in [25]). This implies that, by assuming that processes have access to mutual exclusion modules, we are not assuming anything more than asynchronous shared-memory systems with regular registers.

Process i can also have other local variables. However, other than the sharing of $diningState_i$, $mutex_i$, and $?\Diamond P_i^1$ variables just described, the local variables of i are private to the process.

We now proceed in more detail. We have the following kinds of steps, which are assumed to occur instantaneously:

- a process crash: $crash_i$ for each $i \in \Pi$
- an update to the failure detector variable at a process: $?\Diamond P_i^1$ is set to true or false, $i \in \Pi$
- an update to the *diningState* variable of a process by the dining user: $diningState_i$ is set to "hungry" or "exiting" for each $i \in \Pi$
- an update to a *mutex* variable of a process by the corresponding mutual exclusion module: $X.mutex_i$ is set to "critical" or "thinking" by X for each $i \in \Pi$
- process $i \in \Pi$ executes some code

A step that is the execution of some code by process i must be in one of the following formats:

1. changes to local variables only
2. changes to local variables followed by one invocation (read or write(v)) of an operation on a shared register
3. one response for an operation (return(v) or ack) on a shared register followed by changes to local variables

In the first case, the code is executed only if a certain predicate on i's state, called a *guard*, is true. The guard, together with the code, is called a *guarded command*. In the second case, the code is also executed only if a guard is true. Shortly (in the definition of an execution) we will require that the next step by i after a step of case 2 must be a step of case 3. That is, these two consecutive steps must consist of the invocation and response of a single shared register operation together with

(optionally) some changes to local variables. In our pseudocode, we represent these two steps as a single guarded command, but when this guarded command is executed, it takes two steps, since operations on shared registers are not instantaneous.

A *(dining) system state* is a vector of process states, one per (dining) process. Note that a system state does not record anything about the internals of the dining user or the mutual exclusion modules (other than what is indicated by the local *mutex* variables of the diner) or anything about the values of the shared registers.

An *execution* consists of an alternating sequence σ of system states and steps, beginning with an (arbitrary) system state that satisfies the following conditions:

- There is at most one crash step per process, and if p_i crashes, then there are no later steps by p_i, $i \in \Pi$. If a crash occurs for i, then i is said to be *faulty*, otherwise it is *correct*.
- For each diner $i \in \Pi$, code steps by i that invoke a shared register operation or contain a response to a shared register operation come in pairs (invoking step first, responding step second), and the only other step by i that can come in between is a crash. (Note that each operation response must be preceded by an invocation for that operation.)[3]
- The invocations and responses on each shared register R satisfy regularity: After extracting all the invocations and responses for R from all the code steps, the values returned by the reads must satisfy regularity as defined above.
- Unless process $i \in \Pi$ has crashed, every invocation of a shared register operation by process i has a response.
- The failure detector steps, which update the $?\Diamond P^1$ local variables, satisfy the

[3]For each process i, invocations and responses occurring in pairs prevent i from being in a state in which, after a transient failure occurs, i is waiting for a response without having a preceding invocation to the shared register. This is a common assumption for stabilizing algorithms that involve read/write operations on shared registers. (e.g. [25, 34, 38])

specification of $?\Diamond P^1$ given above.

- The dining user steps "preserve dining well-formedness" (from the user's perspective), i.e., for each $i \in \Pi$, after some point, $diningState_i$ is set to "hungry" only if its current value is "thinking", and $diningState_i$ is set to "exiting" only if its current value is "eating".
- For each correct $i \in \Pi$, $diningState_i$ is not forever eating.
- Each mutual exclusion module X is correct. I.e., there is a suffix σ' of σ in which the following are true:
  - The steps by X "preserve mutex well-formedness" (from the implementor's perspective), i.e., for each $i \in \Pi$, $X.mutex_i$ is set to "critical" only if its current value is "trying", and $X.mutex_i$ is set to "remainder" only if its current value is "exiting".
  - If all the processes participating in X are correct, then, $X.mutex_i$ is not forever "exiting" for each process i that is participating in X.
  - Suppose that in some suffix σ'' of σ' all processes i that participate in X "preserve mutex well-formedness" (from the user's perspective), i.e., $X.mutex_i$ is set to "trying" only if its current value is "remainder", and $X.mutex_i$ is set to "exiting" only if its current value is "critical". Then the following are true in some suffix of σ'':
    - If i and j are both correct and both participate in X, then $X.mutex_i$ and $X.mutex_j$ are not both equal to "critical" in any system state.
    - If all the processes participating in X are correct and no process participating in X is critical forever, then any process that is trying in X eventually is critical.
- Each correct $i \in \Pi$ is given infinitely many opportunities to take steps. (See discussion below concerning pseudocode for more details.)

Correctness condition: Our task is to design a distributed algorithm for the (dining) processes in Π such that every execution has a suffix in which the following four properties hold:

- Well-formedness: Each $i \in \Pi$ "preserves dining well-formedness" (from the implementor's perspective), i.e., for all $i \in \Pi$, $diningState_i$ is set to "eating" only if the current value is "hungry", and $diningState_i$ is set to "thinking" only if the current value is "exiting".
- Finite Exiting: For each correct $i \in \Pi$, $diningState_i$ is not forever "exiting" (with respect to dining).
- Exclusion: If i and j are both correct and are neighbors, then $diningState_i$ and $diningState_j$ are not both equal to "eating" in any system state.
- FL1 Liveness: If $i \in \Pi$ is correct and all its neighbors are correct, then if $diningState_i$ is "hungry" in some state, there is a later system state in which $diningState_i$ is "eating".

We say that an *algorithm implements stabilizing failure-locality-1 dining* if every execution of the algorithm has a suffix in which the above correctness condition is satisfied. We use the term "stabilizing" in the following sense: Consider each execution α of any distributed algorithm that satisfies the above correctness condition. There can be a prefix of α in which some of the above four properties are violated (this is because α begins with an arbitrary state). However, it is guaranteed that the four properties are eventually and forever satisfied in α. That is, the system stabilizes to a state s in α such that the execution starting at state s satisfies the four properties.

Here is an explanation for how our pseudocode maps to this model of executions. Pseudocode is presented as a set of guarded commands. If a guard is continuously true, then eventually the corresponding command is executed. Each command in-

cludes at most one shared register operation. If a command includes a shared register operation, then this is actually two (instantaneous) steps: the first step ends with the invocation of the operation, and the second step begins with the response of the operation. If a command does not include a shared register operation, then it corresponds to a single step.

3.4 ADW-based Stabilizing Dining

In this section, we present a stabilizing failure-locality-1 dining algorithm that is based on the Asynchronous Doorway (ADW) algorithm in [13] and the concept of skepticism in [63]. In the original ADW algorithm, each process shares a single token called a *fork* with each of its neighbors. For a hungry process i to eat, it must first enter the *doorway* by obtaining permission from all of its neighbors through a ping/ack protocol. Only after process i enters the doorway, it requests for the missing forks. Also, while i is inside the doorway, i does not give its neighbors permissions to enter the doorway. The hungry process i can start to eat if it is both inside the doorway and possesses all forks shared between itself and its neighbors. After eating, i satisfies all deferred requests and exits the doorway. In our algorithm, we simulate both the ping/ack and fork activities using multiple mutual exclusion modules.

Our algorithm also uses the concept of skepticism to satisfy the FL1 Liveness condition: a process i becomes "skeptical" if and only if $?\Diamond P_i^1$ is true and, as long as p is skeptical, p satisfies all requests from its neighbors by going or remaining outside the doorway.

3.4.1 Algorithm Description

Let N_i be the neighbor set of process i. For each pair of neighboring processes i and j, we use two mutual exclusion modules to simulate the ping-ack activity in entering the doorway; i and j are the only processes that participate in these two

modules. The mutual exclusion module that simulates the activities of i sending the ping and $j \in N_i$ replying with an ack is denoted as $Doorway^{(i,j)}$ (note that the superscript is an ordered pair). $Doorway^{(i,j)}.mutex_i = critical$ indicates that process i obtained permission to enter the doorway from process j.

We also use mutual exclusion modules to simulate fork activities. For each pair of neighboring processes i and j, the mutual exclusion module that is being used to simulate a unique fork shared by i and j is denoted as $Fork^{\{i,j\}}$ (note that the superscript is an unordered pair). $Fork^{\{i,j\}}.mutex_i = critical$ indicates that the fork shared between i and j is at process i.

3.4.1.1 Variables

Each process i has a local variable $indoors_i \in \{T, F\}$. Each pair of neighboring processes i and j share two single-writer-single-reader (SWSR) regular registers $Req^{(i,j)}$ and $Req^{(j,i)}$ where the domain of each register is $\{T, F\}$; the first and second element of the ordered pair on the superscript indicates the single writer and the single reader, respectively. For each pair of neighboring processes i and j, process i writes T to register $Req^{(i,j)}$ to tell process j that it needs the fork. For each process $j \in N_i$, process i has a local variable $localReq_i^{(j,i)} \in \{T, F\}$ that stores the most recent value that is read from register $Req^{(j,i)}$.

3.4.1.2 Actions

The pseudocode of the actions is given in Figures 3.1 and 3.2, and described next.

For each correct process i, Action $D.1$ is always enabled. When Action $D.1$ is executed with respect to $j \in N_i$, i checks if j is requesting the fork by performing a read operation on $Req^{(j,i)}$. If i reads T from $Req^{(j,i)}$, and if i is outside the doorway, or inside the doorway but has lower id than j, then i releases the fork by setting $Fork^{\{i.j\}}.mutex_i$ to *exiting*. Action $D.2$ is enabled when i's dining

⟨ **Local Variables and Shared Registers** ⟩

1: local constant N_i; // neighbor set of i (incorruptible)
2: local variable $diningState_i \in \{thinking, hungry, eating, exiting\}$;
3: local variable $?\Diamond P_i^1 \in \{T, F\}$;
4: local variable $indoors_i \in \{T, F\}$;
5: $\forall j \in N_i$: local variables $localReq_j^{(j,i)} \in \{T, F\}$;
6: $\forall j \in N_i$: local variables $Doorway^{(i,j)}.mutex_i \in \{remainder, trying, critical, exiting\}$;
7: $\forall j \in N_i$: local variables $Doorway^{(j,i)}.mutex_i \in \{remainder, trying, critical, exiting\}$;
8: $\forall j \in N_i$: local variables $Fork^{\{i,j\}}.mutex_i \in \{remainder, trying, critical, exiting\}$;
9: $\forall j \in N_i$: SWSR registers $Req^{(i,j)}, Req^{(j,i)} \in \{T, F\}$;

⟨ **Program Actions** ⟩

10: $\{j \in N_i\} \rightarrow$ Action $D.1$
11: read $localReq_i^{(j,i)} \leftarrow Req^{(j,i)}$;
12: **if** $localReq_i^{(j,i)} \wedge Fork^{\{i,j\}}.mutex_i = critical \wedge$
$(\neg indoors_i \vee (diningState_i = hungry \wedge indoors_i \wedge i < j))$ **then**
13: $Fork^{\{i,j\}}.mutex_i \leftarrow exiting$;

14: $\{diningState_i = thinking$ Action $D.2$
$\vee diningState_i = exiting\} \rightarrow$
15: $indoors_i \leftarrow F$;
16: **for all** $j \in N_i$ **do**
17: **if** $Fork^{\{i,j\}}.mutex_i = critical$ **then**
18: $Fork^{\{i,j\}}.mutex_i \leftarrow exiting$;
19: **if** $Doorway^{(j,i)}.mutex_i = critical$ **then**
20: $Doorway^{(j,i)}.mutex_i \leftarrow exiting$;
21: $diningState_i \leftarrow thinking$;

22: $\{j \in N_i \wedge \neg indoors_i\} \rightarrow$ Action $D.3$
23: write $Req^{(i,j)} \leftarrow F$;

Figure 3.1: ADW-based stabilizing failure-locality-1 dining algorithm; code for process i (part 1 of 2).

```
24: {diningState_i = hungry ∧ ¬indoors_i} →                          Action D.4
25: for all j ∈ N_i do
26:    if Fork^{i,j}.mutex_i = critical then
27:       Fork^{i,j}.mutex_i ← exiting;
28:    if Doorway^{(j,i)}.mutex_i = critical then
29:       Doorway^{(j,i)}.mutex_i ← exiting;
30: if ¬?◇P_i^1 then
31:    if ∀j ∈ N_i : Doorway^{(i,j)}.mutex_i = critical then
32:       indoors_i ← T;
33:       for all r ∈ N_i do
34:          Doorway^{(i,r)}.mutex_i ← exiting;
35:    else
36:       for all j ∈ N_i do
37:          if Doorway^{(i,j)}.mutex_i = remainder then
38:             Doorway^{(i,j)}.mutex_i ← trying;

39: {j ∈ N_i ∧ diningState_i = hungry                                Action D.5
    ∧indoors_i} →
40: if Doorway^{(j,i)}.mutex_i = remainder then
41:    Doorway^{(j,i)}.mutex_i ← trying;
42: if Fork^{i,j}.mutex_i = remainder then
43:    Fork^{i,j}.mutex_i ← trying;
44: if Fork^{i,j}.mutex_i = trying then
45:    write Req^{(i,j)} ← T;

46: {diningState_i = hungry ∧ indoors_i                              Action D.6
    ∧(∀j ∈ N_i : Fork^{i,j}.mutex_i = critical)} →
47: diningState_i ← eating;

48: {?◇P_i^1 ∧ diningState_i ≠ eating} →                             Action D.7
49: indoors_i ← F;
```

Figure 3.2: ADW-based stabilizing failure-locality-1 dining algorithm; code for process i (part 2 of 2).

state is either *thinking* or *exiting*. By executing Action $D.2$, i exits the doorway ($indoors_i = F$) and releases unnecessary resources that i was holding (by setting both $Fork^{\{i,j\}}.mutex_i$ and $Doorway^{(j,i)}.mutex_i$ to *exiting*), and changes its dining state to *thinking*. Action $D.3$ is enabled when i is outside the doorway and when this action is executed with respect to $j \in N_i$, i informs j that it does not need the fork by setting $Req^{(i,j)}$ to F.

When process i is hungry, i enables different actions depending on whether i is inside or outside the doorway. Action $D.4$ is enabled when i is hungry and outside the doorway. Upon executing Action $D.4$, process i first releases unnecessary resources (by setting both $Fork^{\{i,j\}}.mutex_i$ and $Doorway^{(j,i)}.mutex_i$ to *exiting*) and then if i obtained permission to enter the doorway from all of its neighbors ($\forall j \in N_i$: $Doorway^{(i,j)}.mutex_i = critical$), then it enters the doorway by setting $indoors_i$ to T and immediately sets $Doorway^{(i,j)}.mutex_i$ to *exiting* for all $j \in N_i$. If i has not yet obtained permission to enter the doorway from $j \in N_i$ ($Doorway^{(i,j)}.mutex_i \neq critical$), then i asks for the permission by setting $Doorway^{(i,j)}.mutex_i$ to *trying*. Note that we enforce the concept of skepticism by allowing process i to enter the doorway only when i is not skeptical.

Action $D.5$ is enabled when i is hungry and inside the doorway. When Action $D.5$ is executed with respect to $j \in N_i$, if i does not have the fork shared between i and j, i requests for the fork by setting $Fork^{\{i,j\}}.mutex_i$ to *trying* and by informing j that it needs the fork (by writing T to $Req^{(i,j)}$). Process i also tries to enter the critical section with respect to module $Doorway^{(j,i)}$. If i satisfies $Doorway^{(j,i)}.mutex_i = critical$, then j cannot enter its doorway; this prevents j from eating an infinite number of times while i is continuously hungry.

Action $D.6$ is enabled when i is hungry, inside the doorway, and possesses all forks shared between itself and all its neighbors. Process i simply starts to eat when

this action is executed.

Finally, Action $D.7$ implements the concept of skepticism: if process i is not eating and $?\Diamond P_i^1$ suspects that there is a crashed neighbor ($?\Diamond P_i^1 = T$), then i goes outside the doorway.

3.4.2 Proof Outline

Here, we provide an outline of the proof. The complete proof of correctness is presented in Section 3.4.3.

We first identify two stable predicates to assist our proofs (Lemmas 3.4.1–3.4.5). The predicates are: for each correct process i, (1) if i is thinking, then i is outside the doorway, and (2) if i is eating, then i is inside the doorway and it holds all forks shared between itself and all of its neighbors.

The Well-formedness property and Finite Exiting property directly follows from the pseudocode (Lemmas 3.4.6 and 3.4.8). Also, from the pseudocode, we immediately observe that mutex well-formedness is preserved from the user's perspective (Lemma 3.4.7). This implies that the mutual exclusion modules used in our algorithm are correct (we can utilize the safety and progress properties of the mutual exclusion modules for our proofs).

The Exclusion property (Lemma 3.4.9) is shown using the second stable predicate (explained above) and the safety property of mutual exclusion modules: there is an infinite suffix of any arbitrary execution of the algorithm in which, (1) a unique fork is shared between each process (since forks are modeled as mutual exclusion modules) and (2) if a correct process i eats, then i holds all forks shared between itself and all of its neighbors.

The FL1 Liveness property (Lemma 3.4.13) is shown in two steps. In the first step, we show that each correct hungry process that is at least two hops away from

any crashed process and is inside the doorway eventually eats. Specifically, we first prove that if there exists a correct hungry process i that is at least two hops away from any crashed process and is inside the doorway but never eats, then i must have a correct hungry neighboring process j such that j is at least two hops away from any crashed process, is inside the doorway, and $i < j$ (Lemma 3.4.10); here, the property of the $?\Diamond P^1$ is used to show the existence of such process j. Then, we derive a contradiction using the total ordering of node ids (Lemma 3.4.11). The proofs utilize the first stable predicate (explained above), and the safety/progress property of both modules $Doorway^{(\cdot)}$ and $Fork^{\{\cdot\}}$.

In the second step (Lemma 3.4.12), we show that each correct hungry process i that is at least two hops away from any crashed process and is outside the doorway eventually enters the doorway. The proof utilizes the progress property of modules $Doorway^{(i,j)}$ for each $j \in N_i$.

3.4.3 Proof of Correctness

We first define the set of "diner safe" states for each process in the system, and then we prove that the predicates that define the diner safe states are stable predicates (predicates that eventually become true and remain true thereafter).

Diner Safe States. Process i is said to be in a *diner safe state* if $((diningState_i = thinking) \rightarrow \neg indoors_i) \wedge ((diningState_i = eating) \rightarrow (indoors_i \wedge (\forall j \in N_i : Fork^{\{i,j\}}.mutex_i = critical)))$ is true. The system is said to be in a diner safe state if and only if every live process (a process that has not yet crashed) is in a diner safe state.

Fix σ to be an arbitrary execution of the algorithm in Figures 3.1 and 3.2, and let s be an arbitrary state in σ. We first show that, for each correct process i, predicate $((diningState_i = thinking) \rightarrow \neg indoors_i) \wedge ((diningState_i = eating) \rightarrow$

$(indoors_i \wedge (\forall j \in N_i : Fork^{\{i,j\}}.mutex_i = critical)))$ is a stable predicate in σ.

Lemma 3.4.1. *For any process i, if $(diningState_i = thinking) \rightarrow \neg indoors_i$ is true in state s, then it remains true for each state s' after s.*

Proof. The proof is straightforward from the fact that (1) $diningState_i$ is set to *thinking*, only when variable $indoors_i$ is set to F by Action $D.2$, and (2) variable $indoors_i$ is set to T only when $diningState_i$ is *hungry* (Action $D.4$). □

Lemma 3.4.2. *For any process i, if $(diningState_i = eating) \rightarrow (indoors_i \wedge (\forall j \in N_i : Fork^{\{i,j\}}.mutex_i = critical))$ is true in state s, then it remains true for each state s' after s.*

Proof. The proof is straightforward from the fact that (1) the precondition to set $diningState_i$ to *eating* is $indoors_i = T$ and for all $j \in N_i$, $Fork^{\{i,j\}}.mutex_i = critical$ by Action $D.6$, (2) variable $indoors_i$ is set to F only when p is not *eating* (Actions $D.2$ and $D.7$), and (3) for any $j \in N_i$, variable $Fork^{\{i,j\}}.mutex_i$ is set to *exiting* or *trying* only when i is not *eating* or $indoors_i$ is F (Actions $D.1$, $D.2$ and $D.4$). □

Lemma 3.4.3. *For each correct process i, if the system is in a state s where $(diningState_i = thinking) \rightarrow \neg indoors_i$ is false, then there exists a state s' after s such that $(diningState_i = thinking) \rightarrow \neg indoors_i$ is true in s'.*

Proof. The lemma immediately holds if process i becomes *hungry*, *critical*, or *exiting*. Thus, we only need to consider the case when i is *thinking* in all states after s. In this case, Action $D.2$ is enabled in s and remains enabled until executed. Thus, Action $D.2$ is eventually executed at i and upon being executed, variable $indoors_i$ is set to F. □

Lemma 3.4.4. *For each correct process i, if the system is in a state s where $(diningState_i = eating) \rightarrow (indoors_i \wedge (\forall j \in N_i : Fork^{\{i,j\}}.mutex_i = critical))$*

is false, then there exists a state s' after s such that $(diningState_i = eating) \rightarrow (indoors_i \wedge (\forall j \in N_i : Fork^{\{i,j\}}.mutex_i = critical))$ is true in s'.

Proof. Since each correct process does not eat forever in σ, there exists a state s' after s in which $diningState_i$ is not *eating*. Hence, in state s', $(diningState_p = eating) \rightarrow (indoors_p \wedge (\forall q \in N_p : Fork^{\{p,q\}}.mutex_p = critical))$ is true. □

From Lemmas 3.4.1, 3.4.2, 3.4.3, and 3.4.4, we directly get:

Lemma 3.4.5. *The system reaches a diner safe state and remains in a diner safe state thereafter.*

Now, we prove the four correctness conditions described in Section 3.3. We start by showing the Well-formedness condition.

Lemma 3.4.6. *There exists a suffix of σ in which Well-formedness is satisfied.*

Proof. By the definition of execution and by Lemma 3.4.5, there exists an infinite suffix σ' of σ that (1) begins in a diner safe state and remains in a diner safe state forever, and (2) dining well-formedness is preserved from the user's perspective. Considering suffix σ', the proof is straightforward from the fact that, for each process i, (1) $diningState_i$ is set to *eating* only if the current value of $diningState_i$ is *hungry* (Action $D.6$), and (2) $diningState_i$ is set to *thinking* only if the current value of $diningState_i$ is either *exiting* or *thinking* (Action $D.2$). □

To ensure that each mutual exclusion module X eventually satisfies the safety and progress condition mentioned in Section 3.3, we show that X eventually preserves mutex well-formedness:

Lemma 3.4.7. *For each correct process i, there exists a suffix of σ in which each mutual exclusion module that i participates in preserves mutex well-formedness.*

Proof. By the definition of execution and by Lemma 3.4.5, there exists an infinite suffix σ' of σ that (1) begins in a diner safe state and remains in a diner safe state forever, and (2) mutex well-formedness is preserved from the implementor's perspective. For the proof, we only consider suffix σ' of σ.

For each $j \in N_i$, there are three mutual exclusion modules that i participates in: $Doorway^{(i,j)}$, $Doorway^{(j,i)}$, and $Fork^{\{i,j\}}$. It is straightforward by Actions $D.2$, $D.4$, and $D.5$ that both $Doorway^{(i,j)}$ and $Doorway^{(j,i)}$ preserve mutex well-formedness from the user's perspective. We can also directly see from Actions $D.1$, $D.2$, $D.4$, and $D.5$ that $Fork^{\{i,j\}}$ preserves mutex well-formedness from the user's perspective. □

By Lemmas 3.4.5, 3.4.6, and 3.4.7, there exists an infinite suffix σ' of σ that (1) begins in a diner safe state and remains in a diner safe state forever, (2) dining well-formedness is preserved from both the user's and implementor's perspective, and (3) each mutual exclusion module preserves mutex well-formedness from both the user's and implementor's perspective. In addition, by Lemma 3.4.7, there exists a suffix σ'' of σ' in which each mutual exclusion module satisfies the safety and progress conditions described in Section 3.3. For the remaining proofs, we only consider suffix σ'' of σ.

Since Action $D.2$ enforces each process i that satisfies $diningState_i = exiting$ to change its dining state to *thinking*, we immediately get the following lemma:

Lemma 3.4.8. *σ'' satisfies Finite Exiting.*

Lemma 3.4.9. *There exists a suffix of σ'' in which Exclusion is satisfied.*

Proof. Since each correct process does not eat forever in σ'', there exists a suffix σ_1 of σ'' in which every correct process that eats *starts* eating in σ_1.

Suppose, in contradiction, that in some state of σ_1, there exist two correct neighboring processes i and j that eat concurrently. First note that, in σ_1, processes can

only start to eat by executing Action $D.6$. By Lemma 3.4.2, as long as i (resp. j) is eating, variable $Fork^{\{i,j\}}.mutex_i$ (resp. $Fork^{\{i,j\}}.mutex_j$) remains as *critical*. This implies that both $Fork^{\{i,j\}}.mutex_i$ and $Fork^{\{i,j\}}.mutex_j$ are set to *critical* while i and j are eating concurrently which contradicts the safety condition of mutual exclusion modules. □

We prove FL1 Liveness in two parts: First we show that, for each process i that is correct and does not have any crashed neighbors, if i is *hungry* and inside the doorway ($indoors_i = T$), then i eventually eats. Then, we show that, for each process i that is correct and does not have any crashed neighbors, if i is *hungry* and outside the doorway ($indoors_i = F$), then i eventually enters the doorway ($indoors_i = T$).

Lemma 3.4.10. *Suppose some process i is correct and does not have any crashed neighbors in σ''. Also, suppose i is hungry and inside the doorway ($indoors_i = T$) but never eats in σ''. Then, there exists a process $j \in N_i$ such that in an infinite suffix of σ'', j satisfies*

(a) $diningState_j = hungry$,

(b) $indoors_j = T$

(c) j does not have any crashed neighbors,

(d) $i < j$, *and*

(e) $Fork^{\{i,j\}}.mutex_j = critical$.

Proof. Since all nodes in N_i are correct, there exists an infinite suffix σ_1 of σ'' in which $?\Diamond P_i^1$ is false. For this proof, we only consider suffix σ_1.

Note that since i never eats, Action $D.5$ is always enabled and all other actions, except $D.1$ and $D.7$, are always disabled at i in σ_1. Also, note that since Action $D.5$ is always enabled at i in σ_1, for all $r \in N_i$, if i is not yet in the critical section with

respect to module $Fork^{\{i,r\}}$, then process p always tries to enter the critical section with respect to module $Fork^{\{i,r\}}$ by setting $Fork^{\{i,r\}}.mutex_i$ to *trying*.

Claim. No process in N_i eats infinitely often in σ_1.

Proof. Suppose, in contradiction, there exists a process $r \in N_i$ that eats infinitely often in σ_1. Since Lemma 3.4.6 enforces r to execute Action $D.2$ infinitely often (in order to transit from $diningState_r = eating$ to $diningState_r = thinking$, Action $D.2$ must be executed), r cannot satisfy $Doorway^{(r,i)}.mutex_r = critical$ in an infinite suffix of σ_1. Thus, by the progress condition of mutual exclusion modules, i satisfies $Doorway^{(r,i)}.mutex_i = critical$ in an infinite suffix of σ_1. This means that, at r, the condition on line 31 of Action $D.4$ will eventually and forever evaluate to false which in turn prevents r from enabling Action $D.6$ since variable $indoors_r$ cannot be set to T after setting it to F through Action $D.2$. A contradiction. □

The above claim implies that every process in N_i is stuck in some dining state. Since no correct process eats forever, processes in N_i cannot be eating eventually forever in σ_1. Also, by Lemma 3.4.8, each process $r \in N_i$ cannot satisfy $diningState_r = exiting$ in an infinite suffix of σ_1. Thus, every process in N_i must be *thinking* or *hungry* in an infinite suffix of σ_1.

Suppose every process in N_i is either *thinking*, or *hungry* and outside the doorway eventually forever in σ_1. In this case, at i, Action $D.6$ will be enabled until it is executed since for any $r \in N_i$ (1) $diningState_r = thinking$ implies that $indoors_r = F$ by Lemma 3.4.1, (2) if $indoors_r = F$ holds eventually forever, then process r eventually and forever writes F to register $Req^{(r,i)}$ by Action $D.3$, (3) process p eventually sets $localReq_i^{(r,i)}$ to F by Action $D.1$ which in turn evaluates the condition on line 12 of Action $D.1$ to false eventually forever, (4) if $diningState_r = hungry$ and $indoors_r = F$ holds eventually forever, then Action $D.4$ ensures that process r exits

its critical section with respect to module $Fork^{\{i,r\}}$, and (5) process r will eventually never set $Fork^{\{i,r\}}.mutex_r$ to *trying* because Action $D.5$ is disabled eventually forever at r. However, this directly contradicts the assumption that i never eats in σ_1.

Thus, there exists a process $r' \in N_i$ that is eventually hungry forever in σ_1, however, r' could be either inside the doorway eventually forever in σ_1, or entering and exiting the doorway infinitely often in σ_1. By the code, the only case that r' can exit the doorway without changing its dining state is when $?\Diamond P^1_{r'}$ becomes true. Since $?\Diamond P^1_{r'}$ is guaranteed to eventually stabilize, r' cannot enter and exit the doorway infinitely often in σ_1. In addition, if each process $r'' \in N_i$ that is eventually hungry in σ_1 has a crashed neighbor, then $?\Diamond P^1_{r''}$ ensures that Action $D.7$ is enabled and the condition on line 30 of Action $D.4$ evaluates to false eventually forever. In this case, each process r'' becomes *hungry* and outside the doorway eventually forever in σ_1 which again implies that, at i, Action $D.6$ will be enabled until it is executed. Thus, there must exist a process in N_i that is hungry and inside the doorway in an infinite suffix of σ_1 and does not have any crashed neighbors.

Now we show that among the processes in N_i that are hungry and inside the doorway in an infinite suffix of σ_1 and do not have any crashed neighbors, there exists a process that has a greater id than i. Suppose, in contradiction, that for each process $r \in N_p$ that is hungry and inside the doorway in an infinite suffix of σ'' and does not have any crashed neighbors, we have $r < i$. In this case, if $Fork^{\{i,r\}}.mutex_i$ is not yet *critical*, then the execution of Action $D.5$ at i ensures that eventually and forever, T is written to register $Req^{(i,r)}$. Since Action $D.1$ is always enabled at r, r reads T from $Req^{(i,r)}$ in an infinite suffix of σ_1. This implies that, at r, the condition on line 12 of Action $D.1$ evaluates to true eventually and forever which in turn tells us that whenever $Fork^{\{i,r\}}.mutex_r = critical$ holds at r, then $Fork^{\{i,r\}}.mutex_r$ will

eventually be set to *exiting*. On the other hand, at i, the condition on line 12 of Action $D.1$ will evaluate to false in an infinite suffix of σ_1 which implies that if i ever satisfies $Fork^{\{i,r\}}.mutex_i = critical$ in σ_1, then for an infinite suffix of σ_1, i satisfies $Fork^{\{i,r\}}.mutex_i = critical$. Since process i continuously tries to enter the critical section with respect to module $Fork^{\{i,r\}}$ (line 43 of Action $D.5$) and by the progress condition of module $Fork^{\{i,r\}}$, variable $Fork^{\{i,r\}}.mutex_i$ is eventually and forever set to *critical* in σ_1. This causes Action $D.6$ to be enabled in infinite suffix of σ_1 which contradicts the assumption that i does not eat in σ_1.

Finally, we show that for each process $r \in N_i$ that is hungry and inside the doorway in an infinite suffix of σ_1, does not have any crashed neighbors, and has a greater id than i, process r eventually and forever satisfies $Fork^{\{i,r\}}.mutex_r = critical$. Since both i and r are eventually and forever hungry and inside the doorway, Actions $D.1$ and $D.5$ are always enabled at both i and r in an infinite suffix of σ_1. In this case, both i and r eventually and forever reads T from $Req^{(r,i)}$ and $Req^{(i,r)}$, respectively. Thus, the condition on line 12 of Action $D.1$ ensures that in an infinite suffix of σ_1, $Fork^{\{i,r\}}.mutex_i$ is eventually set to *exiting* whenever $Fork^{\{i,r\}}.mutex_i = critical$ holds and the progress condition of module $Fork^{\{i,r\}}$ guarantees that r eventually sets $Fork^{\{i,r\}}.mutex_r$ to *critical*. After setting $Fork^{\{i,r\}}.mutex_r$ to *critical*, process r cannot set $Fork^{\{i,r\}}.mutex_r$ to *exiting* since the condition on line 12 of Action $D.1$ evaluates to false ($r > i$). Therefore, the lemma holds. □

Lemma 3.4.11. *Consider each process i that is correct and does not have any crashed neighbors in σ''. If i is hungry and inside the doorway ($indoors_i = T$), then i eventually eats in σ''.*

Proof. Suppose, in contradiction, that there exists a non-empty set U of processes that are correct and do not have any crashed neighbors such that for all processes

$u \in U$, u is hungry and inside the doorway but never eats in an infinite suffix of σ''. Using U, we construct the directed "waits-for" graph $W = (U, E_W)$ where vertices are processes in U and (u, q) is in E_W if and only if $u < q$ and $Fork^{\{u,q\}}.mutex_q = critical$ holds in an infinite suffix of σ''. By Lemma 3.4.10, each vertex in W has at least one outgoing edge, and thus there is a cycle in W (basic fact from graph theory). This contradicts the total ordering of node ids. □

Lemma 3.4.12. *Consider each process i that is correct and does not have any crashed neighbors in σ''. If i is hungry and outside the doorway ($indoors_i = F$), then i eventually enters the doorway ($indoors_i = T$) in σ''.*

Proof. Since all nodes in N_i are correct, there exists an infinite suffix σ_1 of σ'' in which $?\Diamond P_i^1$ is false. For this proof, we only consider suffix σ_1.

Suppose, in contradiction, that i never enters the doorway. By Action $D.4$, we notice that i enters the doorway when $Doorway^{(i,j)}.mutex_i = critical$ for all $j \in N_i$. Also note that, for any $j \in N_i$, the only case that i exits the critical section with respect to module $Doorway^{(i,j)}$ is when $Doorway^{(i,r)}.mutex_i = critical$ for all $r \in N_i$ (Action $D.4$). Thus, there must exist a process $r' \in N_i$ that satisfies $Doorway^{(i,r')}.mutex_{r'} = critical$ in an infinite suffix of σ_1 because otherwise the progress condition of module $Doorway^{(i,r')}$ ensures that $Doorway^{(i,r')}.mutex_i$ to be set to *critical* in an infinite suffix of σ_1. This implies that both Actions $D.2$ and $D.4$ are executed only finitely many times at r' in σ_1. In this case, r' must satisfy either $diningState_{r'} = thinking$, or $diningState_{r'} = hungry$ and $indoors_{r'} = F$ in an infinite suffix of σ_1 because otherwise Lemma 3.4.11 and the fact that no process eats forever in σ_1 ensures that Action $D.2$ is executed infinitely often. However, if r' satisfies $diningState_{r'} = thinking$ in an infinite suffix of σ_1, then Action $D.2$ is executed infinitely often in σ_1. Also, if r' satisfies $diningState_{r'} = hungry$ and

$indoors_{r'} = F$ in an infinite suffix of σ_1, then Action $D.4$ is executed infinitely often in σ_1. A contradiction. □

From Lemmas 3.4.11 and 3.4.12, we directly get FL1 Liveness:

Lemma 3.4.13. *σ'' satisfies FL1 Liveness.*

We showed that any execution satisfies Well-formedness, Finite Exiting, Exclusion, and FL1 Liveness through Lemmas 3.4.6, 3.4.8, 3.4.9, and 3.4.13, respectively. Therefore, we obtain the following theorem:

Theorem 3.4.14. *The algorithm in Figures 3.1 and 3.2 implements stabilizing failure-locality-1 dining.*

4. NEIGHBOR KNOWLEDGE OF MOBILE NODES IN A ROAD NETWORK

In this section, we provide a solution for nodes to maintain neighbor knowledge where nodes communicate via wireless broadcast and are restricted to move on a two-dimensional road network; a road network is roughly a collection of lines that may intersect each other (see Section 4.3 for details). We can view vehicular ad hoc networks as nodes (vehicles) with wireless communication capabilities moving on a road network. Our solution is an extension of the one-dimensional case in [27] and [70] to two-dimensional space.

Our solution provides *deterministic* guarantees for maintaining neighbor knowledge. Our algorithm divides each line of the road network into segments and utilizes time division multiplexing to avoid wireless broadcast interference and collisions. Assuming that nodes are sufficiently close together, know their future trajectory, and have initial neighbor knowledge, certain nodes (called leader nodes) are periodically identified for each segment. Leader nodes are the only nodes that broadcast neighbor information; restricting wireless broadcasts to be performed only by leader nodes eliminates the risk of collisions within each segment. For a given time unit, leader nodes broadcast neighbor information (including trajectories) following a deterministic collision-free broadcast schedule. This broadcast schedule ensures that simultaneous broadcasts performed by two different leader nodes do not interfere with each other. All nodes update their neighbor knowledge for the next time unit based on the information received from leader nodes. Other applications may utilize the neighbor information provided by our solution to perform their own tasks. We

Part of this section is reprinted from the following paper: ©2012 IEEE. Reprinted, with permission, from Hyun Chul Chung, Saira Viqar, and Jennifer L. Welch, "Neighbor knowledge of mobile nodes in a road network," In proceedings of the 32nd IEEE International Conference on Distributed Computing Systems (ICDCS), June 2012, pages 486-495.

also provide a correctness proof for our solution.

In addition to maintaining neighbor knowledge, we show how fast a message can travel in the network using our broadcast schedule. We first consider message propagation on a single line and then extend the result to the road network. Specifically, given any two points A and B on the road network, we provide a lower bound on the speed of a message initially broadcast at point A in reaching point B.

We also consider grouping nodes on the same line into clusters where a node density requirement only holds within each cluster. Under certain conditions, we show that neighbor knowledge is maintained within a cluster that is formed by merging two clusters on the same line. Furthermore, we show that neighbor knowledge is maintained within a cluster even after a single node merges into that cluster through an intersection.

Our solution relies on nodes having initial neighbor knowledge. We address the issue of obtaining initial neighbor knowledge based on a gossiping algorithm in [30]. We also discuss how to relax the assumption that every node knows its entire future trajectory and provide practical values for the parameters used in this section.

4.1 Contributions

We present a deterministic solution for maintaining neighbor knowledge which guarantees that each node in a two dimensional road network knows its neighboring nodes at all times. This solution can serve as a black box which provides neighbor information to other applications.

Considering the deterministic collision-free broadcast schedule presented as part of our solution in maintaining neighbor knowledge, we determine how fast a message can propagate on a two-dimensional road network by providing a lower bound on the speed of message propagation.

Finally, we consider dynamic clusters of nodes moving on a road network and show, under certain conditions, how neighbor knowledge can be maintained even when clusters merge with each other.

4.2 Related Work

Our algorithm is a generalization of the algorithm of Ellen et al. [27] and Subramanian [70]. Their algorithm applies to nodes that are moving along a one-dimensional line (for example vehicles moving along a single highway). They assume that this one-dimensional line is divided into segments of equal size. These segments are in turn partitioned into a finite number of colors. Segments of the same color are spaced sufficiently apart so that nodes which occupy two different segments of the same color can transmit simultaneously without message collisions. With the help of this coloring of segments a collision-free schedule is designed, which allows nodes to transmit messages, depending on their current location. However, their algorithm cannot be applied to two-dimensional VANETs, with multiple adjacent lanes, or intersecting roads. A natural, and non-trivial extension of their work is to deal with the case of nodes approaching each other on intersections. We provide an algorithm for nodes moving on a road network in which lines may intersect at any angle. In addition to this, we provide an analysis of clusters of nodes gaining knowledge of each other while merging.

In [73], the authors consider nodes moving on a two-dimensional plane. They show how their communication schedule can provide neighborhood knowledge. They do have some discussion about how different types of schedules may help in the dissemination of information along a path of connected nodes on the plane. In their work, they do not provide a characterization of the speed of message propagation achieved by the schedule between two points on the plane. There is also no analysis

of clusters of nodes merging on the plane.

In [19], a reliable neighbor discovery layer for mobile ad hoc networks is defined. Two distributed algorithms are presented which implement this layer with varying progress guarantees. However, these algorithms are implemented on top of a Medium Access Control (MAC) Layer which is specified in [45]. This MAC Layer provides upper bounds on the time for message delivery thereby abstracting away the lower level details of collision detection, contention and scheduling. We adopt a more integrated approach. Our algorithm handles contention and message collisions by means of a deterministic collision-free broadcast schedule.

In [53] and [54], the authors show lower bounds for gossip in the case of static nodes on a one-dimensional line. Different classes of algorithms are defined: singleton algorithms in which only one node can transmit during one time-step, collision free algorithms in which multiple nodes can transmit simultaneously as long as it does not result in a message collision, and unrestricted algorithms in which no restrictions are placed on the nodes. The authors show different lower bounds for the three cases, in terms of the worst-case number of transmission slots required for all-to-all communication (or gossip). Also, in [54], a gossiping algorithm is presented for mobile ad hoc radio networks which uses the schedule defined in [27]. Bounds on the speed of information propagation are presented in a setting where nodes move on a one-dimensional line. In contrast to this, we analyze the worst case time required for a message broadcast by a node p located at point A to reach point B on a two-dimensional road network using our broadcast schedule.

4.3 System Model

We define a *road* network as a collection of one-dimensional lines that may intersect each other where, for any two lines, there is no restriction on the intersecting

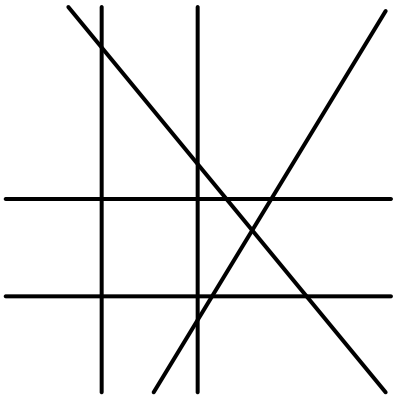


Figure 4.1: A road network. Notice that the angle between two intersecting lines need not be a right angle.

angle; see Figure 4.1. We consider a fixed road network in two-dimensional Euclidean space where nodes move along the lines of the road network with a speed of at most σ. We assume that nodes have unique ids and do not crash.

Nodes communicate via wireless broadcast. We consider a common communication radius R and a common interference radius R' ($\geq R$): node p correctly receives node q's broadcast message if (1) q was within distance R of p during the entire broadcast, and (2) there was no other node broadcasting a message within distance R' of p at any time while q was broadcasting its message.

Each one-dimensional line on the road network is divided into segments of size G. The segmentation of each individual line is independent from each other; there are no restrictions such as intersection points should correspond to segment boundaries, etc. Consecutive segments on a line are numbered in increasing order and colored with m colors $0, 1, ..., m-1$: segment s has a color of s mod m.

Nodes have synchronized clocks which allows nodes to utilize time division multiplexing. Time is divided into *rounds* where each round is further divided into u unit length *time slots*. Broadcasts occur at the beginning of a time slot and the duration

of a time slot is long enough for a node to broadcast a message by the end of the time slot. Similar to segments on a line, consecutive rounds are numbered in increasing order and colored with m colors $0, 1, ..., m-1$: round r has a color of r mod m. We assume that three or more colors are used to color segments and rounds; $m \geq 3$.

We assume that nodes initially know their future trajectory; each node p knows its trajectory function $f_p(t)$ where, given a time t, it returns the exact location of p on the road network. The trajectory function of a node may be exchanged among nodes.

Throughout Section 4, we use the term *Manhattan distance* as follows: Manhattan distance from point A to point B on a road network as the length of the shortest path from A to B *along* the road network. If we just say "distance", then it refers to Euclidean distance. For two points A and B on the road network, we denote $dist_E(A, B)$ and $dist_M(A, B)$ as the Euclidean and Manhattan distance between point A and B on the road network, respectively. Also, for a given time instant and for two nodes p and q, we denote $dist_E(p, q)$ and $dist_M(p, q)$ as the Euclidean and Manhattan distance between p and q on the road network, respectively.

4.4 A Deterministic Broadcast Schedule

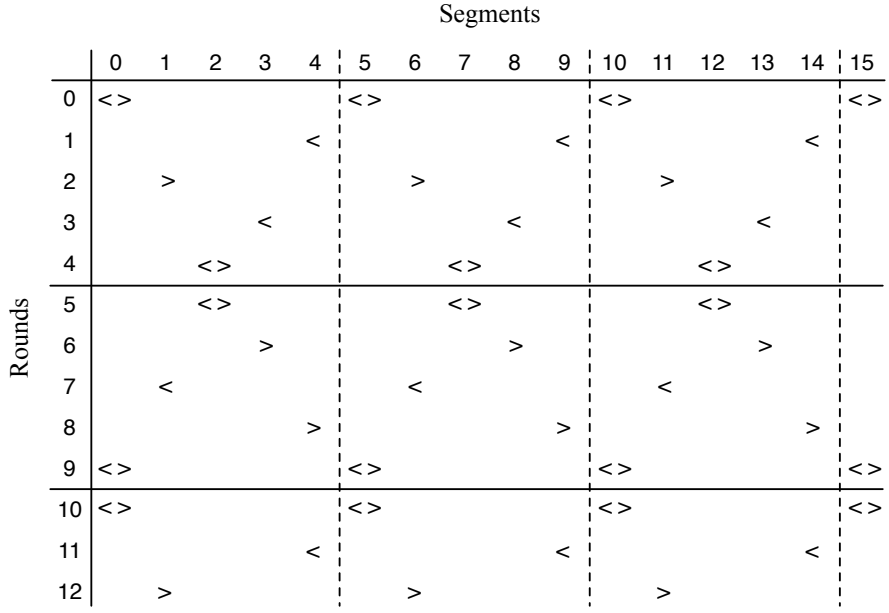
Our broadcast schedule is a small modification of that presented in [27] and [70]. Each m consecutive rounds, starting from round 0, makes up a *phase*[1]. Each line ℓ is assigned only a single time slot during each round for nodes on ℓ to broadcast messages. We say that two lines ℓ_1 and ℓ_2 are *nearby* if there exists two points x_1 and x_2 on ℓ_1 and ℓ_2, respectively, such that the Euclidean distance between x_1 and x_2 is less than $R + R'$. If two lines that are nearby use the same time slot in each round,

[1]In [27] and [70], not all segments get a chance to broadcast in a phase since a phase consists of $m-1$ rounds. This causes difficulties in the analysis since the case of a node not being able to broadcast in a phase has to be considered. In our schedule, every segment broadcasts in a phase which eases the analysis.

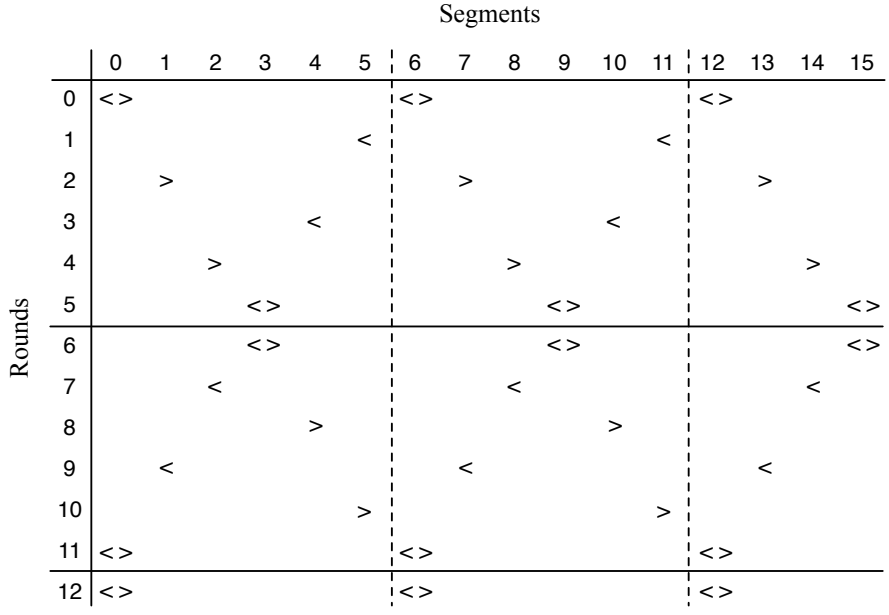
then broadcasts performed on those lines may interfere with each other. We assume that a digital map is initially provided to all nodes in the network which allows each node to know which line it is on and which time slot is assigned to that line. We further assume that the time slot for each line on the digital map is assigned such that any two lines that are nearby use different time slots in each round. This implies that u, the number of time slots per round, is sufficiently large to handle inter-line interference.

For each phase, exactly those nodes that are in segments of color c at the beginning of the phase are allowed to broadcast during a round that has a color of c within that phase. Notice that multiple nodes can be located in the same segment at the beginning of a phase and these nodes have a potential of causing broadcast interference. To avoid interference, nodes in segment s at the beginning of a phase elect a leader node which is the only node that can broadcast on behalf of segment s in that phase. Leader election at the beginning of a phase can be done by local computation under a condition that every node in a segment at the beginning of a phase know the trajectory function of one another in that segment (e.g. by selecting the node with the lowest ID in a segment as the leader).

Our broadcast schedule is illustrated in Figure 4.2 (the figure shows two cases: when $m = 5$ and when $m = 6$). Each row represents a round and each column represents a segment. A "<" (named *leftward* broadcast) indicates a broadcast that is in favor of propagating a message towards segments of decreasing order, a ">" (named *rightward* broadcast) indicates a broadcast that is in favor of message propagation towards segments of increasing order, and a broadcast that is favorable to both directions is indicated by "<>". In our schedule, (1) every segment gets a chance to broadcast in every phase, (2) the segment that performed the last broadcast in phase π again performs the first broadcast in phase $\pi + 1$, (3) the first and last



(a) When $m = 5$.



(b) When $m = 6$.

Figure 4.2: Broadcast schedule examples.

broadcast of each phase are both "<>", (4) the second broadcast of phase $\pi + 1$ is ">" (resp. "<") if the second last broadcast of phase π was "<" (resp. ">"), and (5) leftward and rightward broadcasts within a phase are interleaved. Each one-dimensional line in the road network follows this schedule. The drawback of our schedule is that there is a delay in message propagation on phase boundaries since the same segment broadcasts in two consecutive rounds. However, as m increases, phase boundary delays occur less frequently which in turn means that phase boundary delays become less influential in message propagation.

4.5 Assumptions and Constraints on Parameters

We have handled intra-segment and inter-line interference. However, the broadcast schedule is still exposed to inter-segment interference on a line; since segments of the same color on the same line broadcast simultaneously, interference might occur if those segments are not sufficiently far apart. To take care of inter-segment interference, we introduce the first constraint:

$$mG - 2mu\sigma \geq R + R' \tag{4.1}$$

where mG corresponds to the distance between two consecutive segments of the same color and $mu\sigma$ is the maximum distance a node can move during a phase. Constraint 4.1 tells us that the broadcasts of two leader nodes, one representing segment s and the other representing segment $s + m$ on the same line, do not interfere with each other even if the two nodes approach each other in maximum speed during a phase.

The following constraint tells us that we only allow nodes to cross at most one

segment boundary in a phase:

$$mu\sigma < G \tag{4.2}$$

We assume a density requirement on the network: on each line, there exists at least one node in every interval of length L at all times. We consider the following constraint on L:

$$L \leq (R - 3G - 8mu\sigma)/2 \tag{4.3}$$

We assume that the Manhattan distance between any two intersection points is greater than $L + 2G + 6mu\sigma$. This assumption enforces a node in between two consecutive intersections A and B to remain in between A and B for a certain number of phases.

We also assume that for each line on the road network, the distance between an endpoint of the line and its nearest intersection is at least $L + 2G + 6mu\sigma$.

In addition, we assume that each node p starts at the beginning of phase 0 and possesses trajectory functions of other nodes within a distance of $R + 2mu\sigma$ from p at the beginning of phase 0. We say that node p *knows* node q if p possesses q's trajectory function.

4.6 Maintaining Neighbor Knowledge

For simplicity, we assume that, when a leader node broadcasts, it broadcasts all trajectory functions that it knows. This assumption may be relaxed, as in [27] and [70], by making each node p to only keep trajectory functions of other nodes that are nearby p (if p has the trajectory function of q, then p can decide whether or not to dispose q's trajectory function since p can calculate its distance to q at any given

time). We now show that nodes maintain neighbor knowledge with the broadcast schedule. More specifically, we show that each node p knows every other node that is within Manhattan distance of R from itself at any given time (Theorem 4.6.2). To do this, we first show that neighbor knowledge is maintained at the beginning of each phase:

Lemma 4.6.1. *For all phases π,*

(a) if $\pi > 1$, then broadcast collisions do not occur during phase $\pi - 1$, and

(b) for all nodes p and q, if $dist_M(p, q) \leq R + 2mu\sigma$ at the beginning of phase π, then p knows q at the beginning of phase π.

Proof. We use induction on π. For the basis ($\pi = 0$), part (a) is vacuously true and part (b) follows by our initialization assumptions.

For the inductive step, we assume that the lemma holds for phase π and we will show that it holds for phase $\pi + 1$.

We first show part (a). Since we have assumed that any two lines that are nearby use different time slots in each round, it is only required to show that collisions do not occur within a single line during phase π.

The inductive hypothesis implies that for all nodes p and q, if $dist_M(p, q) \leq G$ at the beginning of phase π, then p knows q at the beginning of phase π since

$$\begin{aligned} R + 2mu\sigma &\geq 2L + 3G + 8mu\sigma && \text{(by Constraint 4.3)} \\ &> G. \end{aligned}$$

Thus, at the beginning of phase π, nodes in the same segment know each other and a unique leader can be elected for each non-empty segment. Note that, by

assumption, leaders are the only nodes that are allowed to broadcast on behalf of their corresponding segments.

Due to our broadcast schedule, leaders representing different segments of the same color may broadcast simultaneously during phase π. Also, at the beginning of phase π, the distance between any two leaders that represent different segments of the same color is at least mG. Note that due to the upper bound σ on node speed and the fact that a phase lasts mu time, the maximum distance a node can travel during a phase is $mu\sigma$. Thus, during phase π, for all pair of leader nodes ℓ_1 and ℓ_2 that represent different segments of the same color, we get

$$\begin{aligned} dist_E(\ell_1, \ell_2) &\geq mG - 2mu\sigma \\ &\geq R + R' \qquad \text{(by Constraint 4.1)} \end{aligned}$$

which proves that broadcasts of ℓ_1 and ℓ_2 do not interfere with each other.

Now, we show part (b). First note that since the maximum Manhattan distance a node can travel during a phase is $mu\sigma$, for all nodes p and q, if $dist_M(p, q) \leq R+2mu\sigma$ at the beginning of phase $\pi + 1$, then $dist_M(p, q) \leq R + 4mu\sigma$ at the beginning of phase π. Thus, it is sufficient to show that, during phase π, p learns about every node q such that $dist_M(p, q) \leq R + 4mu\sigma$ at the beginning of phase π (if it does not already know q).

If $dist_M(p, q) \leq R + 2mu\sigma$ at the beginning of phase π, then by the inductive hypothesis, p knows q at the beginning of phase π; thus p knows q at the beginning of phase $\pi + 1$. Now, assume that $dist_M(p, q) > R + 2mu\sigma$ at the beginning of phase π. We will show that there must exist a node ℓ such that

(i) ℓ is the leader of a segment during phase π,

(ii) $dist_M(p, \ell) \leq R - 2mu\sigma$ at the beginning of phase π, i.e., ℓ is within broadcast range of p throughout phase π, and

(iii) $dist_M(\ell, q) \leq R + 2mu\sigma$ at the beginning of phase π, i.e., ℓ knows q at the beginning of phase π, by the inductive hypothesis.

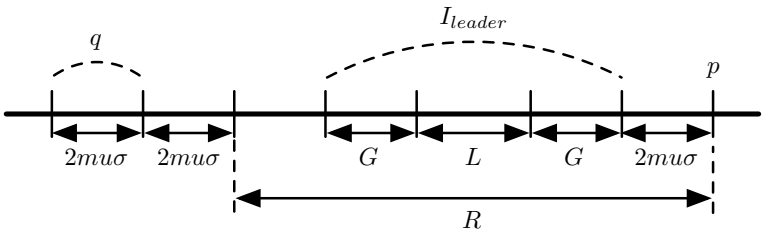
Thus, p learns about q during phase π (if it does not already know q) and p knows q at the beginning of phase $\pi + 1$.

–(*Case* 1) The path from p to q (along the lines of the road network) either consists of a single line or the first line change occurs at distance at least $L/2 + G + 2mu\sigma$: Consider the interval I_{leader} of length $L + 2G$ that resides on the same line as p lies on at the beginning of phase π where (1) one endpoint is at distance $2mu\sigma$ and the other endpoint is at distance $L+2G+2mu\sigma$ from p, and (2) there exists a subinterval of length at least $L/2 + G$ that overlaps with the path from p to q (see Figures 4.3a and 4.3b).

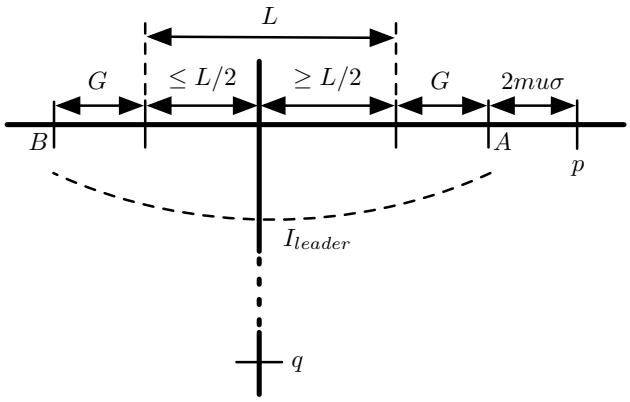
(i) By the density assumption, the subinterval of length L centered inside I_{leader} contains at least one node r. Either r or another node in the same segment as r is a leader for phase π. Since the segment length is G, somewhere in I_{leader} there is a leader node ℓ.

(ii) To show $dist_M(p, \ell) \leq R - 2mu\sigma$ at the beginning of phase π, first note that $dist_M(p, \ell) \leq L + 2G + 2mu\sigma$ by the definition of I_{leader}. Then,

$$\begin{aligned} & R - 2mu\sigma \\ \geq\ & 2L + 3G + 8mu\sigma - 2mu\sigma && \text{(by Constraint 4.3)} \\ \geq\ & L + 2G + 2mu\sigma \\ \geq\ & dist_M(p, \ell) && \text{(by note above)} \end{aligned}$$



(a) Case 1: no line changes occur.



(b) Case 1: a line change occurs.

Figure 4.3: Proof of Lemma 4.6.1 (Case 1).

(iii) To show $dist_M(\ell, q) \leq R + 2mu\sigma$ at the beginning of phase π, we further divide into two cases depending on the location of ℓ with respect to the intersection where the first line change occurs:

—(*Case* 1a) A line change does not occur from p to q (see Figure 4.3a) or ℓ is on the same side of the intersection as p (between point A and the intersection in Figure 4.3b) at the beginning of phase π: In this case, ℓ is located within

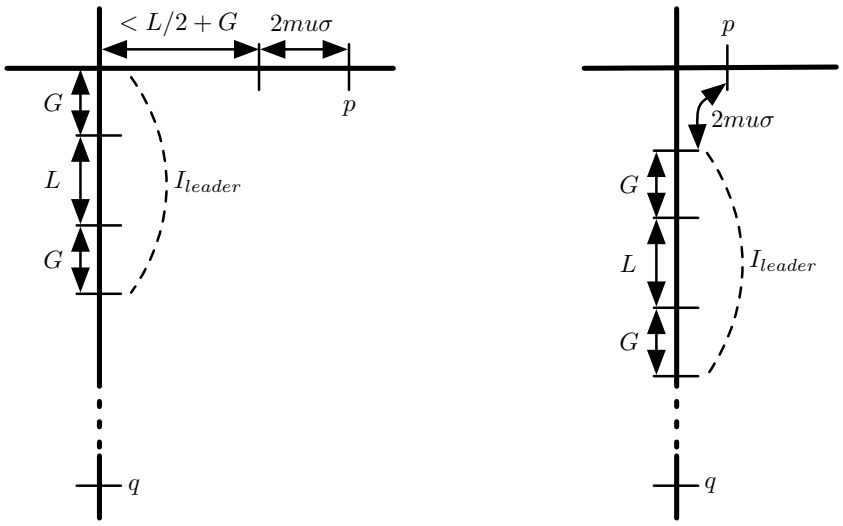
the path from p to q. Thus,

$$\begin{aligned} & dist_M(\ell, q) \\ = \; & dist_M(p, q) - dist_M(p, \ell) \\ \leq \; & R + 4mu\sigma - dist_M(p, \ell) && \text{(by assumption)} \\ \leq \; & R + 4mu\sigma - 2mu\sigma && \text{(by def. of } I_{leader}) \\ = \; & R + 2mu\sigma \end{aligned}$$

—(*Case* 1b) ℓ is on the other side of the intersection as p (between point B and the intersection in Figure 4.3b) at the beginning of phase π: Since at least half of I_{leader} is on the same side of the intersection as p, we get

$$\begin{aligned} & dist_M(\ell, q) \\ \leq \; & dist_M(p, q) - 2mu\sigma \\ \leq \; & R + 4mu\sigma - 2mu\sigma && \text{(by assumption)} \\ = \; & R + 2mu\sigma \end{aligned}$$

–(*Case* 2) The distance between p and the first intersection point on the path to q is at least $2mu\sigma$ but less than $L/2 + G + 2mu\sigma$ (see Figure 4.4a): Consider the interval I_{leader} of length $L + 2G$ that resides on the intersecting line where (1) one endpoint is on the intersection point, and (2) overlaps with the path from p to q. Note that, since the distance between any two intersection points is at least $L + 2G + 6mu\sigma$, no line intersects I_{leader}. The Manhattan distance between p and any point in I_{leader} is at most $3L/2 + 3G + 2mu\sigma$. By Constraint 4.3, it follows that $3L/2 + 3G + 2mu\sigma \leq R$. Thus, I_{leader} is entirely between p and q.



(a) Case 2. (b) Case 3.

Figure 4.4: Proof of Lemma 4.6.1 (Cases 2 and 3).

(i) Same as in (Case 1), there exists a leader node ℓ in I_{leader} at the beginning of phase π.

(ii) To show $dist_M(p, \ell) \leq R - 2mu\sigma$ at the beginning of phase π, note that

$$\begin{aligned} & dist_M(p, \ell) \\ \leq\ & 3L/2 + 3G + 2mu\sigma && (\ell \text{ is in } I_{leader}) \\ \leq\ & R - L/2 - 6mu\sigma && \text{(by Constraint 4.3)} \\ <\ & R - 2mu\sigma \end{aligned}$$

(iii) To show $dist_M(\ell, q) \leq R + 2mu\sigma$ at the beginning of phase π, first note that $dist_M(p, \ell) \geq 2mu\sigma$ since the distance between p and intersection is at least

$2mu\sigma$. Thus,

$$\begin{aligned}
& dist_M(\ell, q) \\
= \; & dist_M(p, q) - dist_M(p, \ell) \\
\leq \; & R + 4mu\sigma - dist_M(p, \ell) && \text{(by assumption)} \\
\leq \; & R + 4mu\sigma - 2mu\sigma && \text{(by note above)} \\
= \; & R + 2mu\sigma
\end{aligned}$$

–(*Case* 3) The distance between p and the first intersection point on the path to q is less than $2mu\sigma$ (see Figure 4.4b): Consider the interval I_{leader} of length $L + 2G$ that (1) resides on the intersecting line where one endpoint is at Manhattan distance $2mu\sigma$ and the other endpoint is at Manhattan distance $L + 2G + 2mu\sigma$ from p, and (2) overlaps with the path from p to q. Note that one endpoint of I_{leader} is at most at distance $2mu\sigma$ from the intersection. Since any two intersections are at least a distance $L + 2G + 6mu\sigma$ apart, no line intersects I_{leader}. The Manhattan distance between p and a point in I_{leader} is at most $L + 2G + 2mu\sigma$. By Constraint 4.3, it follows that $L + 2G + 2mu\sigma \leq R$. Thus, I_{leader} is entirely on the path from p to q. For the rest of the proof, part (i) and (ii) is similar to (Case 1), and part (iii) is similar to (Case 1a). □

Using Lemma 4.6.1, we can show that, for any given node, knowledge about other nodes within Manhattan distance of R can be maintained at all times:

Theorem 4.6.2. *Every node knows every other node that is within a Manhattan distance of R from itself at all times.*

Proof. Since the time duration of each phase is mu and the maximum node speed is bounded by σ, two nodes p and q that are separated by a Manhattan distance

of at most R during phase π can be separated by a Manhattan distance of at most $R + 2mu\sigma$ at the beginning of phase π. Hence, by Lemma 4.6.1, p and q already knew each other at the beginning of phase π. □

4.7 Speed of Message Propagation

For this section, we introduce the following constraint which says that the communication radius is strictly less than the half of the distance between two consecutive segments of the same color on a same line:

$$R < \lfloor m/2 \rfloor G. \tag{4.4}$$

Note that the above constraint does not violate Constraints 4.1 to 4.3.

We analyze how fast messages can travel on the road network using our broadcast schedule. To do so, we first analyze the speed of message propagation on a single line. Specifically, we show that, considering message propagation from left to right (message propagation from right to left will be analogous) on a single line, if a leader node p representing segment s does a broadcast of "$>$" or "$<>$" at round r, then there exists a constant $c > 0$ and a leader node q that represents a segment $s + \Delta$ for some integer $\Delta \geq 0$ such that q performs a broadcast of "$>$" or "$<>$" by round $r + c\Delta$; this will show that the speed of information propagation on a single line is at least one segment per c rounds.

Using the speed of message propagation on a single line, we analyze the speed of information propagation on any given path in the road network. Specifically, given points A and B on the road network where A is the location where some node p broadcasts a message, we will analyze the worst case time required for the message broadcast by p to reach B. The speed of message propagation can be easily calculated

by dividing the $dist_M(A, B)$ with the total time required for a message to reach B from A. The basic idea of our analysis is as follows: We calculate the worst case time δ_1 required to reach point B from point A pretending that A and B lies on a single line. Then, we calculate the worst case time delay δ_2 caused by changing from one line to another along the path from A to B by comparing it to the case when that line change has not occurred. If a number of k line changes occurred along the path from A to B, then the worst case total time of message propagation from A to B would be at most $\delta_1 + k\delta_2$.

Before we delve into the analysis of the speed of information propagation, we first make several observations on our schedule that are useful for our analysis. All observations consider message propagation on a single line.

Observation 4.7.1. *Suppose segment s is* not *scheduled to broadcast at the last round of phase π. Then,*

(a) if s did a broadcast of "<>" in round r of π, then segment $s-1$ (resp. $s+1$) does a leftward (resp. rightward) broadcast in round $r+1$ or $r+2$,

(b) if s did a leftward broadcast ("<") in round r of π, then segment $s-1$ does a broadcast of "<>" or "<" in round $r+1$ or $r+2$, and

(c) if s did a rightward broadcast ("<") in round r of π, then segment $s+1$ does a broadcast of "<>" or ">" in round $r+1$ or $r+2$

Observation 4.7.1 basically tells us that, *within a phase*, the difference of broadcast rounds of two adjacent segments is at most 2 assuming a single direction of message propagation.

Observation 4.7.2. *Suppose segment s is scheduled to broadcast at the last round r of phase π. Then, segment $s-1$ (resp. $s+1$) broadcasts in round $r+2$ or $r+3$.*

Observation 4.7.2, along with Observation 4.7.1, shows that the difference of broadcast rounds of two adjacent segments is at most 3 assuming a single direction of message propagation.

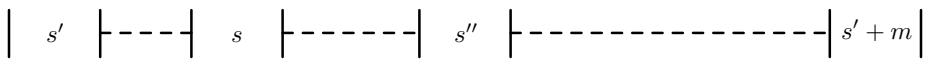
Observation 4.7.3. *Suppose segment s is scheduled to do a broadcast of "<" (resp. ">") in phase π. Then, in phase $\pi+1$, segment $s-1$ is scheduled to do a broadcast of ">" (resp. "<") or "<>", segment s is scheduled to do a broadcast of ">" (resp. "<"), and segment $s+1$ is scheduled to do a broadcast of ">" (resp. "<") or "<>".*

Observation 4.7.3 says that for any three consecutive segments $s-1$, s, and $s+1$ where s does a leftward (resp. rightward) broadcast in phase π, all three segments do a broadcast in favor of propagating the message towards segments of increasing (resp. decreasing) order in phase $\pi+1$.

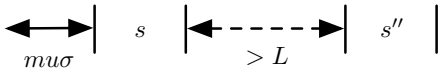
Using the above observations, Lemma 4.7.4 provides the speed of message propagation on a single line. We use a similar approach as in [27] and [70], however, our analysis turns out to be simpler since, in our case, every segment broadcasts in every phase.

Lemma 4.7.4. *The speed of message propagation on a single line is at least one segment per three rounds.*

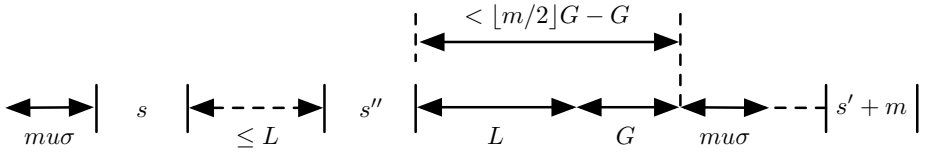
Proof. We prove with the assistance of Figure 4.5. Suppose node p, which is the leader node of segment s at phase π, broadcasts a message m during phase π. We only consider message propagation from left to right where node p performs a broadcast of either ">" or "<>" during phase π (considering "<" will be analogous). Let s' be the nearest segment on the left of s where it is scheduled to broadcast in the first round of phase π. Also, let s'' be the nearest segment on the right of s where it is scheduled to broadcast in the first round of phase $\pi+1$ (See Figure 4.5a). Note that



(a) The construction.



(b) Case 1.



(c) Case 2.

Figure 4.5: Proof of Lemma 4.7.4.

segment s itself can be the segment to broadcast in the first round of phase π or the segment to broadcast in the last round of phase π.

We divide into two cases depending on the distance between segments s and s'': –(*Case* 1) The distance between s and s'' is greater than L (Figure 4.5b): In this case, by the density requirement, there exists a leader node q that represents segment $s + i$ at the beginning of phase π where $s + i \in [s + 1, s'' - 1]$.

We show that q receives message m from p during phase π. The maximum distance between p and q at the beginning of phase π is $L + G$. Even though p and q move away from each other in maximum speed, the distance between p and q will

be at most $L+G+2mu\sigma$ during phase π. Hence, at any given time during phase π,

$$\begin{aligned} dist_E(p,q) &\leq L+G+2mu\sigma \\ &\leq R-L-2G-6mu\sigma && \text{(by Constraint 4.3)} \\ &< R \end{aligned}$$

which implies that q receives m.

Since s'' is the nearest segment to the right of s that broadcasts in the first round of phase $\pi+1$, q performs a rightward broadcast in phase π after p's broadcast due to the definition of our schedule. Hence, by Observation 4.7.1, if p broadcasts at round r, then q does a rightward broadcast by round $r+2i$.

–(*Case* 2) The distance between s and s'' is at most L (Figure 4.5c): In this case, there might not be a node in between segments s and s''. Consider the interval I_{leader} of length $L+G$ from the right endpoint of segment s'' towards segment $s'+m$. First note that, due to the definition of our schedule, the distance between s'' and $s'+m$ (also the distance between s'' and s') is at least $\lfloor m/2 \rfloor G - G$. We get

$$\begin{aligned} \lfloor m/2 \rfloor G - G &> R-G && \text{(by Constraint 4.4)} \\ &\geq 2L+2G+8mu\sigma && \text{(by Constraint 4.3)} \\ &> L+G \end{aligned}$$

which implies that I_{leader} is entirely in between s'' and $s'+m$.

By the density requirement, there exists a node q' in I_{leader} at the beginning of phase $\pi+1$. Let $s+i$ be the segment where node q' lies on at the beginning of phase $\pi+1$ where $s+i \in [s'', s'+m-1]$. Since the leader node of $s+i$ at the beginning of phase π can be at most at distance G from q', there exists a leader node q in I_{leader}

that represents segment $s + i$ at the beginning of phase $\pi + 1$.

We show that q receives p's broadcast message m in phase π. Since q is in I_{leader} at the beginning of phase $\pi + 1$, q could be a distance of at most $mu\sigma$ away from the right endpoint of I_{leader} during phase π. Node p can be a distance of at most $mu\sigma$ away from the left endpoint of segment s. Hence, nodes p and q can be a distance of at most $2L + 3G + 2mu\sigma$ away from each other during phase π. Thus, at any given time during phase π,

$$\begin{aligned} dist_E(p, q) &\leq 2L + 3G + 2mu\sigma \\ &\leq R - 6mu\sigma && \text{(by Constraint 4.3)} \\ &< R \end{aligned}$$

which implies that q receives p's broadcast during phase π.

By the definition of our schedule, q performs a rightward broadcast during phase $\pi + 1$. Also, since s'' is the node that performs the first broadcast of phase $\pi + 1$, s'' is also the segment that performs the last broadcast of phase π. Hence, by Observations 4.7.1 and 4.7.2, if p broadcast at round r, then q does a rightward broadcast by round $r + 3i$.

Combining the results of (Case 1) and (Case 2), we obtain that the speed of information propagation is at least one segment per three rounds. □

The following lemma is useful in determining the time delay of changing the direction of message propagation on a single line which in turn plays a key role in determining the speed of message propagation.

Lemma 4.7.5. *Let I be an interval of length $L + 2G + 6mu\sigma$ on a single line where no other line intersects it and suppose that every node in I knows message m at some*

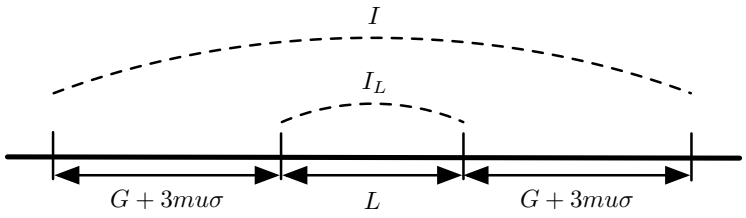


Figure 4.6: Proof of Lemma 4.7.5.

time t during phase π. Then,

(a) there exists a leader node that represents a segment in I such that it knows m and performs a broadcast of ">" or "<>" by the end of phase $\pi + 2$, and

(b) there exists a leader node that represents a segment in I such that it knows m and performs a broadcast of "<" or "<>" by the end of phase $\pi + 2$.

Proof. We only prove part (a); part (b) will be analogous. By the density requirement, during phase π, there exists a node p in an interval I_L of length L that is centered inside I; the left (resp. right) end of I_L and the left (resp. right) end of I are a distance of $G+3mu\sigma$ apart (see Figure 4.6). Let s_p^π be the segment where node p was located at the beginning of phase π. At the beginning of phase π, node p could be at most a distance of $mu\sigma$ from the left and right end of I_L (since the maximum distance a node can travel during a phase is bounded by $mu\sigma$) and the leader node q that represents segment s_p^π in phase π can be at most $G + mu\sigma$ from the left and right end of I_L since p and q have to be in the same segment at the beginning of phase π. During phase π, node q can move at most $G + 2mu\sigma$ from the left and right end of I_L which implies that q will be in I all times during phase π. Thus, q knows m at time t. If node q is scheduled to broadcast after time t within phase π and performs a broadcast of ">" or "<>", then we are done. The remaining cases

we need to consider are:

–(*Case* 1) Node q performed a broadcast of "<" after time t within π: Consider phase $\pi + 1$. Let $s_p^{\pi+1}$ be the segment where node p is located at the beginning of phase $\pi+1$. At the beginning of phase $\pi+1$, node p can be at most $mu\sigma$ apart from the left and right end of I_L and the leader node q' that represents segment $s_p^{\pi+1}$ in phase $\pi + 1$ can be at most $G + mu\sigma$ apart from the left and right end of I_L. At all times during phase π, node q' could have been at most $G + 2mu\sigma$ from the the left and right end of I_L since the maximum distance a node can travel during a phase is bounded by $mu\sigma$. Thus, q' was in I at all times during phase π which implies that q' knows m. By Constraint 4.2, node p can be located in either $s_p^\pi - 1$, s_p^π, or $s_p^\pi + 1$ ($s_p^{\pi+1} \in [s_p^\pi - 1, s_p^\pi + 1]$). Hence, by Observation 4.7.3, q' performs a broadcast of ">" in phase $\pi + 1$.

–(*Case* 2) Node q was scheduled to broadcast before time t during phase π: In this case, node q missed its chance to broadcast m during phase π. Consider phase $\pi+1$. At the beginning of phase $\pi + 1$, there exists a node p' in interval I_L (by the density requirement). Let $s_{p'}^{\pi+1}$ be the segment where node p' is located at the beginning of phase $\pi + 1$. The leader node q_1 that represents $s_{p'}^{\pi+1}$ in phase $\pi + 1$ can be located at most G apart from the left and right end of I_L. At all times during phase π, node q_1 could have been at most $G + mu\sigma$ from the left and right end of I_L since the maximum distance a node can travel during a phase is bounded by $mu\sigma$. Thus, q_1 was in I at all times during phase π which implies that q_1 knows m. If q_1 performs a broadcast of ">" or "<>" in phase $\pi + 1$, then we are done.

The remaining is to consider the case of q_1 performing a broadcast of "<" in phase $\pi + 1$. Now, consider phase $\pi + 2$. Let $s_{p'}^{\pi+2}$ be the segment where node p' is located at the beginning of phase $\pi + 2$. Similar to node p and q' in (Case 1), at the beginning of phase $\pi + 2$, node p' can be at most $mu\sigma$ apart from the left and

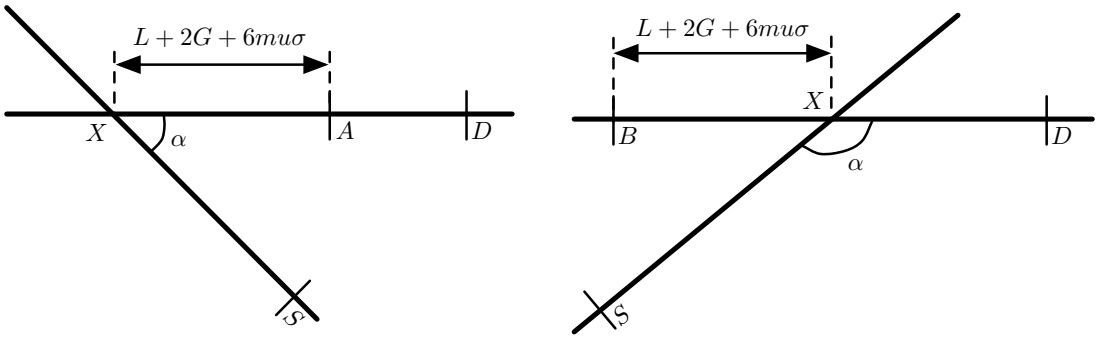
right end of I_L and the leader node q_2 that represents segment $s_p^{\pi+2}$ in phase $\pi+2$ can be at most $G+mu\sigma$ apart from the left and right end of I_L. At all times during phase π, node q_2 could have been at most $G+3mu\sigma$ from the the left and right end of I_L since the maximum distance a node can travel during two phases is bounded by $2mu\sigma$ (note that q_2 is a leader node at the beginning of phase $\pi+2$). Thus, q_2 was in I at all times during phase π which implies that q_2 knows m. The rest of the proof is similar to (Case 1). □

The following simple lemma tells us that in an interval of length $L+2G+2mu\sigma$ on a single line, there always exists a leader node:

Lemma 4.7.6. *Let I be an interval of length $L+2G+2mu\sigma$ on a single line where no other line intersects. Then, during each phase π, there exists a leader node q that remains in I at all times during π.*

Proof. Suppose, in contradiction, that no such leader node exists. By the density requirement, at the beginning phase π, there exists a node p in an interval I_L of length L that is centered inside I; the left (resp. right) end of I_L and the left (resp. right) end of I are a distance of $G+mu\sigma$ apart. Let s_p be the segment where p is located at the beginning of phase π. The leader node q representing segment s_p during phase π can be at most G apart from the left and right end of I_L at the beginning of π since p and q has to be in the same segment at the beginning of π. Since the distance that a node can move during a phase is bounded by $mu\sigma$, q can be at most $G+mu\sigma$ apart from the left and right end of I_L during phase π. Hence, q remains within I at all times during phase π, a contradiction. □

Considering message propagation from any source location S to any destination point D on the road network where there is exactly one line change occurring along



(a) Case 1. (b) Case 2.

Figure 4.7: Proof of Lemma 4.7.7.

the shortest path from S to D, we determine the worst case amount of time message propagation can be delayed due to a line change by comparing it to the case when that line change has not occurred. Given a node p and a position D on the road network, we say that node p, positioned at point S on the road network, performs a *well-directed* broadcast towards position D on the road network if the direction (">", "<", or "<>") of p's broadcast is in favor of the direction of message propagation along the shortest path from S to D on the road network.

Lemma 4.7.7. *Let S and D be positions on the road network where exactly one line change occurs along the shortest path from S to D on the road network. Suppose node p performs a broadcast of message m at point S towards D at time t. Then, message m reaches point D by time $t + 3u(\lceil dist_M(S, D)/G \rceil + \lceil (L + 2G + 6mu\sigma)/G \rceil + 2) + 3u(\lceil (L + 5G + 6mu\sigma)/G \rceil + 1) + 6mu$.*

Proof. We prove with the assistance of Figure 4.7. Let ℓ_1 and ℓ_2 be lines where S and D lies on, respectively. Also, let X be the point on the road network where ℓ_1 and ℓ_2 intersect. We divide into two cases depending on the angle α between ℓ_1 and ℓ_2:

–(*Case* 1) $\alpha \leq \pi/2$ (see Figure 4.7a): Let A be a point on ℓ_2 where $dist_E(A, X) = L + 2G + 6mu\sigma$ and in between X and D. Note that no other line intersects in between X and A since the distance between any two intersection point is greater than $L + 2G + 6mu\sigma$. We further divide into two cases depending on the distance between S and X:

—(*Case* 1a) $dist_E(S, X) \leq L + 2G + 2mu\sigma$: We first show that when p broadcasts message m at time t, every node in between X and A on ℓ_2 receives m. The requirement would be to show that the Euclidean distance between S and A is at most R. $dist_E(S, A)$ becomes maximum when $\alpha = \pi/2$. We can obtain a relationship between $dist_E(S, A)$ and R as the following:

$$\begin{aligned}
& (dist_E(S, A))^2 \\
= \; & (dist_E(X, S))^2 + (dist_E(X, A))^2 \\
= \; & (L + 2G + 2mu\sigma)^2 + (L + 2G + 6mu\sigma)^2 \\
< \; & (2L + 3G + 8mu\sigma)^2 \\
\leq \; & R^2
\end{aligned}$$

Thus, we get $dist_E(S, A) \leq R$.

By Lemma 4.7.5, at most three phases ($3mu$ time) are needed for a leader node q in between point X and point A that possesses m to perform a rightward broadcast towards point D.

Let s_p be the segment on line ℓ_1 where p is located at time t and let E indicate the location of the endpoint of s_p that is closer to S. Now, suppose S and D were on the same single line. In order to directly apply Lemma 4.7.4, the segment s_q on ℓ_2 that q represents should be a segment that is entirely on the path from S to D and at least

a Manhattan distance of G from E. However, s_q might not satisfy these conditions since the segmentation of ℓ_1 and ℓ_2 are independent from each other. Consider an interval I_G of length $3G$ on ℓ_2 that is entirely on the path from S to D where the left endpoint resides on X. Since there is always a segment that is entirely in an interval of length $2G$, there exists a segment s' of ℓ_2 in I_G that is entirely on the path from S to D and at least a Manhattan distance of G from E. Thus, if s_q is a segment that is located on the left of s', we can apply Lemma 4.7.4 on the single line by considering a time delay of which s' being scheduled to perform a rightward broadcast after q performing a rightward broadcast. This time delay can be at most $3u(\lceil 3G/G \rceil + 1) = 12u$ since the speed of message propagation on a single line is one segment per 3 rounds (by Lemma 4.7.4) and there is at most $\lceil 3G/G \rceil + 1$ segments of ℓ_2 that overlap with I_G.

If S and D were on the same line, then, by Lemma 4.7.4, the amount of time required for message m to reach D from S is at most $3u(\lceil dist_M(S, D)/G \rceil + 1)$ since the speed of message propagation on a single line is one segment per 3 rounds (by Lemma 4.7.4) and there can be at most $\lceil dist_M(S, D)/G \rceil + 1$ segments that overlap with with the path from S to D. Hence, the amount of time required for message m to reach D from S on the road network is at most $3u(\lceil dist_M(S, D)/G \rceil + 1) + 3mu + 12u$.

—(*Case* 1b) The distance from S to X is greater than $L+2G+2mu\sigma$: In this case, we need to consider an additional delay that can be caused by p performing a broadcast that is *not* well-directed towards D. Let s_p be the segment on line ℓ_1 where p is located at time t and let I_p be an interval of length $L + 2G + 6mu\sigma$ on ℓ_1 such that p is located in I_p at time t and there is no intersection point in I_p. Note that such I_p exists since the distance between any two intersection points is greater than $L + 2G + 6mu\sigma$. When p broadcasts m at time t, all nodes in I_p receives m since $R > L+2G+6mu\sigma$ by Consttraint 4.3. Now, by applying Lemma 4.7.5, there exists a

leader node q in I_p that performs a well-directed broadcast towards D by time $t+3mu$. The time delay from q performing a well-directed broadcast to s_p being scheduled to perform a well-directed broadcast is at most $3u(\lceil (L+2G+6mu\sigma)/G \rceil + 1)$ since the speed of message propagation on a single line is one segment per 3 rounds (by Lemma 4.7.4) and there is at most $\lceil (L+2G+6mu\sigma)/G \rceil + 1$ segments of ℓ_1 that overlap with I_p.

Let I_ℓ be an interval of length $L+2G+2mu\sigma$ that is in between X and S where one end point lies on X. Message m will make progress of at least one segment per 3 rounds towards the intersection point X (by Lemma 4.7.4) and m will reach a leader node in I_ℓ (there exists a leader node in I_ℓ at all times due to Lemma 4.7.6) which will rebroadcast m at some time t' such that all nodes in between X and A receives m at t'. The rest of the proof is similar to (Case 1a). Hence, the amount of time required for message m to reach D from S on the road network is at most $3u(\lceil dist_M(S,D)/G \rceil + \lceil (L+2G+6mu\sigma)/G \rceil + 2) + 6mu + 12u$.

–(*Case* 2) $\alpha > \pi/2$ (see Figure 4.7b): Let I_ℓ be an interval of length $L+2G+2mu\sigma$ such that one end point lies on X and the other end point is at most a distance of $L+2G+2mu\sigma$ apart from S. Let B be a point on ℓ_2 where $dist_E(B,X) = L+2G+6mu\sigma$ and not in between X and D. Similar to (Case 1), all nodes in between point B and X will receive message m by the broadcast performed by a node in I_ℓ and, by Lemma 4.7.5, at most $3mu$ time will be needed for a leader node q in between point X and point A to perform a rightward broadcast towards point D. Also, similar to (Case 1b), the additional delay of $3mu+3u(\lceil (L+2G+6mu\sigma)/G \rceil+1)$ (which can be caused by p performing a broadcast that is not well-directed towards D) has to be considered.

Let I_G be an interval of length $3G$ on ℓ_2 that is entirely on the path from S to D where the left endpoint resides on X. Also, let s' be a segment of ℓ_2 that is the

rightmost segment among all segments that are entirely in I_G. Now, suppose S and D were on the same single line. Similar to (Case 1a), applying Lemma 4.7.4 requires us to consider an additional time delay of which s' being scheduled to perform a rightward broadcast after q performing a rightward broadcast . This time delay can be at most $3u(\lceil (L + 5G + 6mu\sigma)/G \rceil + 1)$ since the speed of message propagation on a single line is one segment per 3 rounds (by Lemma 4.7.4) and there are at most $\lceil (L + 5G + 6mu\sigma)/G \rceil + 1$ segments of ℓ_2 that overlap with the path from B to the right endpoint of I_G. Hence, the amount of time required for message m to reach D from S on the road network is at most $3u(\lceil dist_M(S, D)/G \rceil + \lceil (L+2G+6mu\sigma)/G \rceil + 2) + 3u(\lceil (L + 5G + 6mu\sigma)/G \rceil + 1) + 6mu$. □

Finally, we determine the worst case amount of time required for a message m broadcast at point S to reach point D where multiple line changes occur along the shortest path from S to D on the road network. In this case, we can consider the shortest path from S to D as a composition of intervals where, in each interval, exactly one line change occurs. Also, we can consider that among all initial broadcasts performed in each such intervals, the only broadcast that cannot be well-formed towards D is the broadcast of m at point S. Hence, we obtain the following theorem:

Theorem 4.7.8. *Let S and D be positions on the road network where k line changes occur along the shortest path from S to D on the road network. Suppose node p performs a broadcast of message m at point S at time t. Then, message m reaches D by time $t + 3u(\lceil dist_M(S, D)/G \rceil + \lceil (L + 2G + 6mu\sigma)/G \rceil + 2) + 3ku(\lceil (L + 5G + 6mu\sigma)/G \rceil + 1) + 3mu(k + 1)$.*

A lower bound on the speed of message propagation from point S to point D on the road network can be easily obtained by dividing $dist_M(S, D)$ with the total time required for a message propagated at point S to reach point D:

Corollary 4.7.9. *Let S and D be positions on the road network where k line changes occur along the shortest path from S to D on the road network. Suppose node p performs a broadcast of message m at point S towards D at time t. Then, the speed of propagating m from S to D is at least* $dist_M(S,D)/(3u(\lceil dist_M(S,D)/G\rceil + \lceil (L + 2G + 6mu\sigma)/G\rceil + 2) + 3ku(\lceil (L + 5G + 6mu\sigma)/G\rceil + 1) + 3mu(k+1))$.

4.8 Dynamic Clusters

In this section, we do not assume that the density requirement holds for the entire road network. Instead, we consider grouping nodes into *clusters* on a single line where the density requirement holds within each cluster. In addition, we assume that nodes move infinitely often and $L \geq G$.

Suppose node p lies on line ℓ at time t. We say that node p is an ℓ-node at t if there exists a time $t' > t$ such that (1) for all times during $[t, t']$, p lies on line ℓ and (2) p moves during $[t, t']$. Since we have assumed that nodes move infinitely often, a node that is located at an intersection point of lines ℓ_1 and ℓ_2 at time t is either an ℓ_1-node or an ℓ_2-node but not both.

Similar to [70], we define a cluster as a group of nodes in which the density requirement holds:

Definition 4.8.1 (L-cluster)**.** *At any given time t, an L-cluster is a maximal set of ℓ-nodes that lie on the same one-dimensional line ℓ such that if we align the ℓ-nodes based on their position on line ℓ, then the distance between any two consecutive ℓ-nodes is at most L.*

Note that each L-cluster is defined on a single one-dimensional line. This means that two nodes that are on different lines cannot be within the same L-cluster even though they are at most a Manhattan distance of L apart from each other.

At any given time, we call the nodes that are located at each end of an L-cluster C as *boundary nodes* of C. We assume that, for each L-cluster, boundary nodes are elected as leaders of their corresponding segment at the beginning of each phase. Again, this could be done by local computation under a condition that every node in a segment at the beginning of a phase know the trajectory function of one another in that segment. Also, note that, at the beginning of each phase, two or more boundary nodes of different L-clusters cannot be in the same segment since we have assumed that $L \geq G$.

We are interested in maintaining neighbor knowledge within an L-cluster C which is formed as a result of (1) merging two L-clusters C_1 and C_2 on the same line where neighbor knowledge was maintained in both C_1 and C_2 before the merge happened, or (2) merging an L-cluster C_1 and a single node p where, before the merge happened, neighbor knowledge was maintained in C_1 and node p was located on a line that is different from the line that C_1 lies on.

As in [70], we define *knowledgeable* L-clusters to formalize the above:

Definition 4.8.2 (Knowledgeable L-cluster)**.** *An L-cluster C, defined on line ℓ is* knowledgeable *during a time interval if (1) it has the same set of ℓ-nodes at every instant of time during the time interval, and (2) at the beginning of each phase during the time interval, every ℓ-node in C knows every other ℓ-node in C that is a distance of at most $R + 2mu\sigma$ from itself.*

Roughly speaking, a knowledgeable L-cluster is a cluster in which the density requirement holds and neighbor knowledge is maintained.

By applying the same proof technique as in (Case 1) of Lemma 4.6.1, we obtain the following simple lemma:

Lemma 4.8.3. *Suppose an L-cluster C, defined on line ℓ, has the same set of ℓ-nodes in every instant of time in $[t, t'']$ and there exists a phase beginning at time $t' \in [t, t'']$ where every ℓ-node in C knows every other ℓ-node in C that is a distance of at most $R + 2mu\sigma$ from itself. Then, C is knowledgeable during $[t', t'']$.*

4.8.1 Merging of Two L-clusters on the Same Line

Maintaining neighbor knowledge after merging two L-clusters has been already discussed in [70]. The analysis given in [70] concentrates on handling the situation of a boundary leader node not being able to broadcast in a phase (this situation may occur since not all segments are scheduled to broadcast in a phase). However, it is missing some details on why a node in one cluster gets to learn about other nodes in a different cluster before the two clusters merge. We provide a rigorous analysis that fills in the missing details.

The following simple lemma shows that if two nodes on the same line are a distance of $R - 3G$ apart, then it takes more than $4mu$ time for those nodes to become a distance of L apart from each other.

Lemma 4.8.4. *Let p and q be two nodes that are on the same line ℓ at time t. Suppose both p and q only move on ℓ and the distance between p and q at time t is $R - 3G$. Then, at time $t + 4mu$, the distance between p and q is greater than L.*

Proof. The time it takes for p and q to become a distance of L apart when both p

and q approach each other in maximum speed is (using Constraint 4.3):

$$\begin{aligned}
& (R - 3G - L)/2\sigma \\
\geq\ & (2L + 3G + 8mu\sigma - 3G - L)/2\sigma && \text{(by Constraint 4.3)} \\
=\ & (L + 8mu\sigma)/2\sigma \\
>\ & 4mu
\end{aligned}$$

Hence, the lemma is proven. □

If a L-cluster is knowledgeable for a sufficiently long period of time, then each node in the cluster not only maintains neighbor knowledge but also "gains" neighbor knowledge within the cluster:

Lemma 4.8.5. *Suppose L-cluster C, defined on line ℓ, is knowledgeable from the beginning of phase π to the beginning of phase $\pi + k$ for some integer $k \geq 0$. And, let S be the set of ℓ-nodes in C from the beginning of phase π to the beginning of phase $\pi + k$. Then,*

(a) at the beginning of phase $\pi + i$ where $i = 0, 1, ..., k$, each node p in S knows every other node in S that is at most a distance of $R + i(L + G + 2mu\sigma) + 2mu\sigma$ from itself, and

(b) at any time during phase $\pi + i$ where $i = 0, 1, ..., k$, each node p in S knows every other node in S that is at most a distance of $R + i(L + G + 2mu\sigma)$ from itself.

Proof. We first show that, assuming part (a) holds, part (b) holds. Assume, at the beginning of phase $\pi + i$, node p in S knows every other node in S that is at most a distance of $R + i(L + G + 2mu\sigma) + 2mu\sigma$ apart from itself. Suppose, in contradiction,

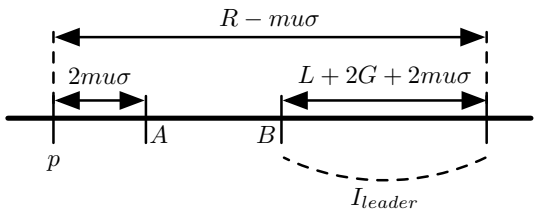


Figure 4.8: Proof of Lemma 4.8.5.

that there exists a node q in S that is at most a distance of $R + i(L + G + 2mu\sigma)$ from p during phase $\pi + i$ but does not know p. Since a node can move at most a distance of $mu\sigma$ in a phase, the distance between p and q at the beginning of phase $\pi + i$ can be at most $R + i(L + G + 2mu\sigma) + 2mu\sigma$. Thus, p must have known q at the beginning of phase $\pi + i$, a contradiction.

Now, we prove part (a) by induction on i. The base case ($i = 0$) immediately follows from the definition of a knowledgeable L-cluster. For the inductive case, we assume the lemma holds for phase $\pi + i$ and we prove it for phase $\pi + i + 1$. Let ℓ be the line where C is defined on. As illustrated in Figure 4.8, consider an interval I_{leader} of length $L + 2G + 2mu\sigma$ to the right of node p at the beginning of phase $\pi + i$ where the right end interval I_{leader} is a distance of $R - mu\sigma$ apart from p (considering I_{leader} to be at the left of p will be analogous). By Lemma 4.7.6, there exists a leader node q' that remains in I_{leader} during phase $\pi + i$. Note that, during phase $\pi + i$, the distance between p and q' can be at most R. Thus, p receives the broadcast of q'during phase $\pi + i$.

By the inductive hypothesis and by the proof in the previous paragraph, leader node q' knows every node that is at most a distance of $R + (L + G + 2mu\sigma)i$ to the right from itself during phase $\pi + i$. Let I_{right} be an interval at the end of phase $\pi + i$ such that p is located at the left end of I_{right} and p knows every node in I_{right}.

We consider the worst case movement of p and q' that minimizes the length of I_{right}. The worst case position of p and q by the end of phase $\pi+i$ would be p being located at a point that is a distance of $mu\sigma$ to the right of p's position at the beginning of phase $\pi+i$ (point A in Figure 4.8) and q being located at the left end of I_{leader} (point B in Figure 4.8). Since p receives the broadcast of q' during phase $\pi+i$ and by Constraint 4.3, p will know every node that is at most a distance of

$$\begin{aligned}
&2R + (L + G + 2mu\sigma)i - (L + 2G + 4mu\sigma) \\
&\geq R + (L + G + 2mu\sigma)i + (L + G + 4mu\sigma) && \text{(by Constraint 4.3)} \\
&= R + (L + G + 2mu\sigma)(i + 1) + 2mu\sigma
\end{aligned}$$

to its right at the end of phase $\pi+i$. □

The following lemma assists in proving the main theorem of this section (Theorem 4.8.7). It is used when there is a need to determine how much information about an L-cluster C_2 is known by a node p in some other L-cluster C_1 during a certain phase. See the proof of Theorem 4.8.7 for details.

Lemma 4.8.6. *Let C be an L-cluster defined on line ℓ. Suppose node p knows every ℓ-node in C that is a distance of at most X from the left (resp. right) end of C at the beginning of phase π where $X \geq 2mu\sigma$. And, let S be the set of ℓ-nodes in C at the beginning of phase π. Then, at all times t during phase π, node p knows every node in S that is a distance of at most $X - 2mu\sigma$ from the leftmost (resp. rightmost) node among the nodes in S at time t.*

Proof. Suppose, in contradiction, that, during phase π, there exists a node q that is in S and is within a distance of $X - 2mu\sigma$ from the leftmost node among the nodes in S but is not known to p. At the beginning of phase π, the distance between

the left end of C and node q must have been greater than X because otherwise p would have already known q. The distance between the left end of C and q becomes minimal when the left boundary (leader) node q' of C and q approach each other in maximum speed during phase π. Even in this case, the distance between the left end of C and q is always greater than $X - 2mu\sigma$ during phase π since both q and q' can move at most a distance of $mu\sigma$ during a phase, a contradiction. □

Now, we show the main theorem of this section which tells us that a new L-cluster C_3, which is formed by merging two knowledgeable L-clusters C_1 and C_2, becomes knowledgeable if C_1 and C_2 were sufficiently far apart before they merged:

Theorem 4.8.7. *Let t'', t, and t' be times such that $t'' < t < t'$. Suppose C_3 is a L-cluster on line ℓ that has the same set of ℓ-nodes in every instant of time during $[t, t']$. Also, suppose that the ℓ-nodes in C_3 are partitioned into sets S_1 and S_2 during $[t'', t)$ such that all nodes in S_1 form a L-cluster C_1 on ℓ and all nodes in S_2 form a L-cluster C_2 on ℓ during $[t'', t)$. Without loss of generality, let C_1 be located to the left of C_2 on ℓ. If (1) C_1 and C_2 are both knowledgeable in $[t'', t)$ and (2) the distance between the right end of C_1 and the left end of C_2 at time t'' is at least $R - 3G$, then C_3 is knowledgeable in $[t, t']$.*

Proof. Let t_π be the most recent time when a phase begins that is strictly less than time t, and let π be the phase that begins at time t_π. We only need to show that, at the beginning of phase $\pi + 1$, every node in C_3 knows every other node in C_3 that is at most a distance of $R + 2mu\sigma$ from itself; Lemma 4.8.3 will take care of the rest of the phase beginnings after phase $\pi + 1$.

Since two clusters C_1 and C_2 were at least a distance of $R - 3G$ apart at time t'', applying Lemma 4.8.4 yields $t - t'' > 4mu$ which in turn implies that, during $[t'', t]$, there exists at least four phase beginnings. So, from the beginning of phase $\pi - 3$ to

the beginning of phase π, both C_1 and C_2 were knowledgeable and, by Lemma 4.8.5, we obtain that for all $i = 0, ..., 3$, (1) at the beginning of phase $\pi - i$, every node in C_1 (resp. C_2) knows every other nodes in C_1 (resp. C_2) that is at most a distance of $R + (L + G + 2mu\sigma)(3 - i) + 2mu\sigma$ from itself, and (2) at any time during phase $\pi - i$, every node in C_1 (resp. C_2) knows every other nodes in C_1 (resp. C_2) that is at most a distance of $R + (L + G + 2mu\sigma)(3 - i)$ from itself. This shows that at the beginning of phase $\pi + 1$, every node that is in S_1 (resp. S_2) knows every other node that is in S_1 (resp. S_2) that is at most a distance of $R + 3(L + G + 2mu\sigma)$ $(> R + 2mu\sigma)$ from itself.

It remains to show that, at the beginning of phase $\pi + 1$, every node that is in S_1 (resp. S_2) knows every other node that is in S_2 (resp. S_1) that is at most a distance of $R + 2mu\sigma$ from itself. Let p be a node in S_1 (considering the case of p being in S_2 will be analogous). We denote the right boundary node of C_1 at the beginning of phase π as p_π and the left boundary node of C_2 at the beginning of phase π as q_π. Let X be the distance between p and p_π at the beginning of phase π. Depending on the value of X, we divide into several cases:

–(*Case* 1) $X \leq R - L - 4mu\sigma$: We first show that p receives q_π's broadcast during phase π. At the beginning of π, the distance between p_π and q_π can be at most $L + 2mu\sigma$. And, during phase π, both p and q_π can move at most a distance of $mu\sigma$ away from each other. Thus, during phase π, the distance between p and q_π can be at most R which implies that p receives q_π's broadcast.

Now, we show that the information contained in q_π's broadcast message is sufficient enough for p to maintain neighbor knowledge. We already know that, during phase π, q_π knows every node in S_2 that is within a distance of $R+3(L+G+2mu\sigma)$ from itself. Since p receives q_π's broadcast during phase π, p will know, at the beginning of phase $\pi+1$, every node in S_2 that is within a distance of $R+3(L+G+2mu\sigma)$

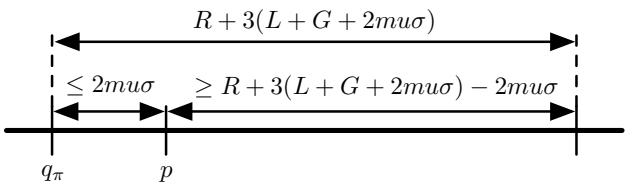


Figure 4.9: Proof of Theorem 4.8.7 (Case 1).

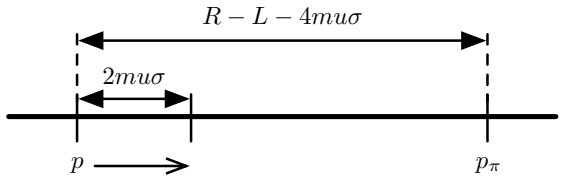
from the leftmost node that is in S_2. In worst case, q_π itself, at the beginning of phase $\pi + 1$, can still be the leftmost node in S_2 by moving a distance of $mu\sigma$ to the left during phase π, and node p could move a distance of $mu\sigma$ to the right causing q_π to be located on the left of p and the distance between q_π and p to be at most $2mu\sigma$. Even in this worst case, p will know every node in S_2 that is within a distance of $R + 3(L + G + 2mu\sigma) - 2mu\sigma$ $(> R + 2mu\sigma)$ to its right at the beginning of phase $\pi + 1$ (see Figure 4.9). Hence, the lemma holds for (Case 1).

–(*Case* 2) $R - L - 4mu\sigma < X \leq R - 2mu\sigma$: We first show that p_π receives $q_{\pi-1}$'s broadcast during phase $\pi - 1$. At the beginning of phase $\pi - 1$, the distance between p_π and $p_{\pi-1}$ can be at most $2mu\sigma$ because otherwise $p_{\pi-1}$ will be located to the right of p_π at the beginning of phase π. Also, at the beginning of phase $\pi - 1$, the distance between $p_{\pi-1}$ and $q_{\pi-1}$ can be at most $L + 4mu\sigma$ since a node in C_1 and a node in C_2 becomes within a distance of L from each other for the first time during phase π (the two nodes that first became within a distance of L from each other during phase π can both move at most a distance of $2mu\sigma$ away from each other in two phases). Since both p_π and $q_{\pi-1}$ can move at most a distance of $mu\sigma$ away from each other during phase $\pi - 1$, the distance between p_π and $q_{\pi-1}$ can be at most $L + 8mu\sigma$ which is less than R by Constraint 4.3. Hence, p_π receives $q_{\pi-1}$'s broadcast during phase $\pi - 1$. Note that, by Lemma 4.8.5, $q_{\pi-1}$ knows every node

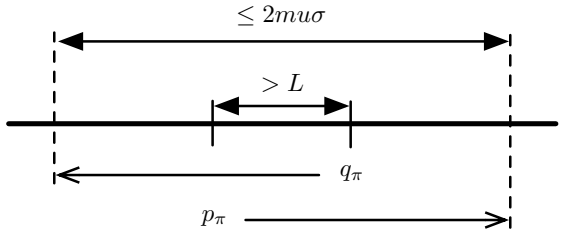
in S_2 that is within a distance of $R+2(L+G+2mu\sigma)$ from the left end of C_2 at all times during phase $\pi-1$ and as a result p_π knows every node in C_2 that is within a distance of $R+2(L+G+2mu\sigma)$ from the left end of C_2 at the beginning of phase π.

The maximum distance between p and p_π is $R-2mu\sigma$ at the beginning of phase π. Both p and p_π can move at most a distance of $mu\sigma$ away from each other during phase π. Thus, the distance between p and p_π can be at most a distance of R apart from each other during phase π which implies that p receives p_π's broadcast during phase π. This means that p can receive $q_{\pi-1}$'s broadcast message indirectly through p_π. Since p_π knows every node in C_2 that is within a distance of $R+2(L+G+2mu\sigma)$ from the left end of C_2 at the beginning of phase π, Lemma 4.8.6 tells us that, at all times t during phases π, p_π knows every node in S_2 that is a distance of at most $R+2(L+G+2mu\sigma)-2mu\sigma$ from the leftmost node in S at time t. Hence, p will know every node in S_2 that is within a distance of $R+2(L+G+2mu\sigma)-2mu\sigma$ from the leftmost node in S_2 at the beginning of phase $\pi+1$ since p receives p_π's broadcast during phase π.

In worst case, node p can move at most a distance of $mu\sigma$ to its right during phase π. Also, in worst case, p_π can still be the rightmost node in S_1 and move a distance of at most $mu\sigma$ to its right, and similarly q_π can still be the leftmost node in S_2 and move a distance of at most $mu\sigma$ during phase π. This may cause q_π to be located on the left of p_π and the distance between q_π and p_π to be at most $2mu\sigma$ in worst case. So, at the beginning of phase $\pi+1$, node p is located on the left of the leftmost node in S_2 (which is q_π in this case) and the distance between p and the leftmost node in S_2 is at least $R-L-7mu\sigma>0$ (See Figure 4.10). Since node p knows every node in S_2 that is at most a distance of $R+2(L+G+2mu\sigma)-2mu\sigma$ ($>R+2mu\sigma$) from the left most node in S_2 at the beginning of phase $\pi+1$ and p



(a) The worst case movement of node p during phase π.



(b) The worst case movement of nodes p_π and q_π during phase π.

Figure 4.10: Proof of Theorem 4.8.7 (Case 2).

is located strictly left to the leftmost node in S_2, the lemma holds for (Case 2).

–(*Case* 3) $R - 2mu\sigma < X \leq R + 4mu\sigma$: By a similar argument as in (Case 2), the distance between node between $p_{\pi-1}$ and $q_{\pi-2}$ is at most $L + 10mu\sigma$. By Constraints 4.2 and 4.3, we get $L + 10mu\sigma < R$. Hence, $p_{\pi-1}$ receives $q_{\pi-2}$'s broadcast message during phase $\pi - 2$. Since, by Lemma 4.8.5, $q_{\pi-2}$ knows every node in S_2 that is within a distance of $R + L + G + 2mu\sigma$ from from the left end of C_2 at all times during phase $\pi - 1$, node $p_{\pi-1}$ knows every node in S_2 that is within a distance of $R + L + G + 2mu\sigma$ from the left end of C_2 at the beginning of phase π.

At the beginning of phase π, consider an interval I_{leader} of length $L + 2G$ where the left end point of I_{leader} is a distance of $R - 2mu\sigma$ to the left of p_π (see Figure 4.11). Since the density requirement holds for C_1, there exists a leader node p' in I_{leader} at the beginning of phase π. We show that p' receives $p_{\pi-1}$'s broadcast message during

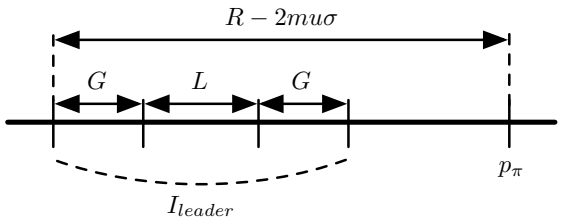


Figure 4.11: Proof of Theorem 4.8.7 (Case 3).

phase $\pi - 1$. Since p_π is the right boundary node of C_1 at the beginning of phase π, either $p_{\pi-1} = p_\pi$ or $p_{\pi-1}$ is located on the left of p_π at the beginning of phase π. Also, the distance between p_π and $p_{\pi-1}$ should be at most $2mu\sigma$ at the beginning of phase π because otherwise $p_{\pi-1}$ could not have been positioned on the right of p_π at the beginning of phase $\pi - 1$. So, the distance between p' and $p_{\pi-1}$ can be at most R during phase $\pi - 1$ since the distance between p' and $p_{\pi-1}$ is at most $R - 2mu\sigma$ at the beginning of phase π and both p' and $p_{\pi-1}$ can move at most a distance of $mu\sigma$ in a phase. Hence, p' receives $p_{\pi-1}$'s broadcast during phase $\pi - 1$. This implies that p' receives $q_{\pi-2}$'s broadcast message indirectly through $p_{\pi-1}$.

The distance between p and p' can be at most $L + 2G + 6mu\sigma$ at the beginning of phase π. Since both p and p' can move at most a distance of $mu\sigma$, the distance between p and p' during phase π can be at most $L+2G+8mu\sigma$. By Constraint 4.3, we get $L + 2G + 8mu\sigma < R$. Hence, p receives the broadcast message of p' during phase π. This implies that p receives $q_{\pi-2}$'s broadcast message indirectly through $p_{\pi-1}$ and p'. By inductively applying Lemma 4.8.6, we obtain that node $p_{\pi-1}$ knows every node in S_2 that is within a distance of $R + L + G - 2mu\sigma$ $(= R + L + G + 2mu\sigma - 4mu\sigma)$ from the leftmost node in S_2 at the beginning of phase $\pi + 1$.

Similar to (Case 2), node p can move at most a distance of $mu\sigma$ to its right during phase π in worst case. Also, in worst case, p_π can still be the rightmost node

in S_1 and move a distance of at most $mu\sigma$ to its right, and similarly q_π can still be the leftmost node in S_2 and move a distance of at most $mu\sigma$ during phase π. This may cause q_π to be located on the left of p_π and the distance between q_π and p_π to be at most $2mu\sigma$ in worst case. So, at the beginning of phase $\pi + 1$, node p is located on the left of the left most in S_2 and the distance between p and the leftmost node in S_2 is at least $R - 5mu\sigma > 0$. It is required for p to only know every node in S_2 that is at most a distance of $7mu\sigma$ $(= R + 2mu\sigma - (R - 5mu\sigma))$ from the leftmost node in S_2 at the beginning of phase $\pi + 1$. By Constraints 4.2 and 4.3, we get $7mu\sigma < R + L + G - 2mu\sigma$. Hence, the lemma holds for (Case 3).

–(*Case* 4) $X > R+4mu\sigma$: In this case, the distance between node p and the leftmost node in S_2 at the beginning of phase $\pi + 1$ is greater than $R + 2mu\sigma$ since nodes can only move at most a distance of $mu\sigma$ during a phase. Hence, during phase π, there is no need for p to learn nodes in S_2. □

4.8.2 Merging of an L-cluster and a Single Node on a Different Line

Since we consider a two-dimensional road network, nodes on line ℓ_1 can merge into an L-cluster on line ℓ_2 through the intersection of ℓ_1 and ℓ_2. In this section, we show how an L-cluster maintains neighbor knowledge even after a single node merges into it through an intersection:

Theorem 4.8.8. *Let t'', t, and t' be times such that $t'' < t < t'$. Suppose C_3 is a L-cluster on line ℓ_1 that has the same set of ℓ_1-nodes in every instant of time during $[t, t']$. Also, suppose that the ℓ_1-nodes in C_3 are partitioned into a set S_1 and a single node p during $[t'', t)$ such that all nodes in S_1 form a L-cluster C_1 on ℓ_1 during $[t'', t)$ and node p is an ℓ_2-node in a L-cluster C_2 on line ℓ_2 during $[t'', t)$ where $\ell_1 \neq \ell_2$. If (1) C_1 and C_2 are knowledgeable in $[t'', t)$ and (2) the Manhattan distance between node p and the intersection of ℓ_1 and ℓ_2 at time t'' is at least $R - 3G - 5mu\sigma$, then*

C_3 is knowledgeable in $[t, t']$.

Proof. Let t_π be the most recent time when a phase begins that is strictly less than time t, and let π be the phase that begins at time t_π. Same as in the proof of Theorem 4.8.7, we only need to show that, at the beginning of phase $\pi + 1$, every node in C_2 knows every other node in C_2 that is at most a distance of $R + 2mu\sigma$ apart from itself (applying Lemma 4.8.3 afterwards proves the theorem).

Since node p was at least a Manhattan distance of $R - 3G - 5mu\sigma$ apart from the intersection point of lines ℓ_1 and ℓ_2 at time t'', the difference between t and t'' is:

$$\begin{aligned} & (R - 3G - 5mu\sigma)/\sigma \\ \geq\ & (2L + 3G + 8mu\sigma - 3G - 4mu\sigma)/\sigma && \text{(by Constraint 4.3)} \\ >\ & 3mu\sigma. \end{aligned}$$

Thus, there are at least three phase beginnings during $[t'', t)$.

By the definition of a L-cluster, at time t, there exists a node q in S_1 where the distance between p and q is at most L. Let q_π and $q_{\pi-1}$ be leader nodes of the segment where q was in at the beginning of phase π and $\pi - 1$, respectively.

We first show that p knows every node in S_1 that is at most a distance of $R+2mu\sigma$ from itself at the beginning of phase $\pi+1$. At the beginning of phase π, the distance between node q and q_π can be at most G since they must be in the same segment. During phase π, both q and q_π can move at most a distance of $mu\sigma$. So, the distance between q and q_π can be at most $G + 2mu\sigma$ during phase π. Since the distance between p and q is at most L at some time (time t) during phase π, the Manhattan distance between p and q during phase π can be at most $L + 2mu\sigma$. Hence, the Manhattan distance between p and q_π during phase π can be at most $L + G + 4mu\sigma$ which implies that p receives q_π's broadcast message during phase π

(since $L+G+4mu\sigma < R$ by Constraint 4.3). By Lemma 4.8.5, node q_π knows every node in S_1 that is at most a distance of $R+2(L+G+2mu\sigma)+2mu\sigma$ from itself at the beginning of phase π and, since p receives $q'_\pi s$ broadcast message during phase π, p will know every node in S_1 that is at most a distance of $R+2(L+G+2mu\sigma)$ from node q_π at all times during phase π. Hence, p will know every node within a distance of $R+2mu\sigma$ at the beginning of phase $\pi+1$ since $R+2(L+G+2mu\sigma)-(L+G+4mu\sigma) > R+2mu\sigma$ (by Constraint 4.2 and the assumption of $L \geq G$).

The remaining is to show that, at the beginning of phase $\pi+1$, every node q' in S_1 that is at most a distance of $R+2mu\sigma$ from node p knows p. Let $p_{\pi-2}$ be the leader node of the segment where node p was in at the beginning of phase $\pi-2$. Since L-cluster C_2 is knowledgeable at the beginning of phase $\pi-2$ and $L \geq G$, node $p_{\pi-2}$ knows p at the beginning of phase $\pi-2$. We show that $q_{\pi-1}$ receives $p_{\pi-2}$'s broadcast message during phase $\pi-2$. Since the Manhattan distance between nodes p and q becomes at most L at some time (time t) during phase π, the Manhattan distance between p and q can be at most $L+6mu\sigma$ during phase $\pi-2$ (a node can move at most a distance of $3mu\sigma$ during three phases). The distance between nodes p and $p_{\pi-2}$ can be at most $G+2mu\sigma$ during phase $\pi-2$ since p and $p_{\pi-2}$ is in the same segment at the beginning of phase $\pi-2$. The distance between nodes q and $q_{\pi-1}$ can be at most $G+2mu\sigma$ during phase $\pi-2$ since q and $q_{\pi-1}$ is in the same segment at the beginning of phase $\pi-1$. Hence, during phase $\pi-2$, the Manhattan distance between $p_{\pi-2}$ and $q_{\pi-1}$ can be at most $L+2G+10mu\sigma < R$ (by Constraint 4.2, Constraint 4.3, and the assumption of $L \geq G$) which implies that $q_{\pi-1}$ receives $p_{\pi-2}$'s broadcast message and gets to know p during phase $\pi-2$.

Since nodes q and $q_{\pi-1}$ are in the same segment at the beginning of phase $\pi-1$, the distance between q and $q_{\pi-1}$ can be at most $G+2mu\sigma$ during phase $\pi-1$. Node $q_{\pi-1}$ is a leader node which broadcasts during phase $\pi-1$. Hence, at the beginning

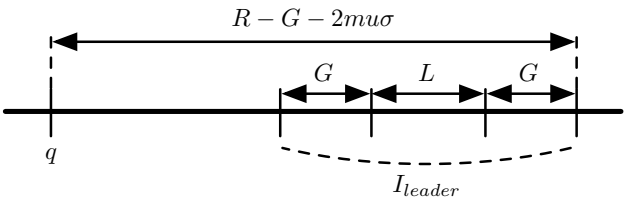
of phase π, every node that is at most a distance of $R - G - 2mu\sigma$ apart from node q will know p by receiving $q_{\pi-1}$'s broadcast message.

Consider an interval I_{leader} of length $L + 2G$ on line ℓ_1 at the beginning of phase π where (1) it is located on the right of node q (considering the case of I_{leader} being located on the left of q will be analogous) and (2) the distance between q and the right end of I_{leader} is $R - G - 2mu\sigma$ (See Figure 4.12a). Suppose, at the beginning of phase π, the distance between q and the rightmost node in S_1 is greater than $R - G - 2mu\sigma$ (otherwise, every node in S_1 that is located on the right of q at the beginning of phase π already knows p). Then, since the density requirement holds in C_1 at the beginning of phase π, there exists a leader node q' in I_{leader} at the beginning of phase π. Also, node q' knows p since it is at most a distance of $R - G - 2mu\sigma$ apart from q at the beginning of phase π. We consider the worst case movement of q and q' during phase π such that the least number of nodes located on the right of q learns about p. The worst case movement will be node q moving a distance of $mu\sigma$ to its right and node q', which is located at the right end of I_{leader} at the beginning of phase π, moving a distance of $mu\sigma$ to its left during phase π. Note that q' broadcasts during phase π since it is a leader node (See Figure 4.12b). Hence, at the beginning of phase $\pi + 1$, every node that is at most a distance of

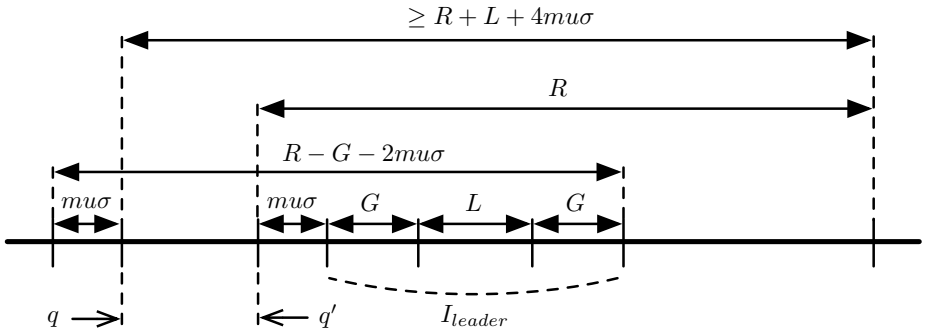
$$\begin{aligned}
& R - G - 2mu\sigma - (L + 2G + 2mu\sigma) + R \\
= \; & 2R - L - 3G - 4mu\sigma \\
\geq \; & R - L - 3G - 4mu\sigma + (2L + 3G + 8mu\sigma) && \text{(by Constraint 4.3)} \\
= \; & R + L + 4mu\sigma
\end{aligned}$$

from q knows p.

We already know that the distance between node p and q at the beginning of phase



(a) Interval I_{leader} (beginning of phase π).



(b) The worst case movement of nodes q and q' during phase π.

Figure 4.12: Proof of Theorem 4.8.8.

$\pi + 1$ can be at most $L + 2mu\sigma$. Since $(R + L + 4mu\sigma) - (L + 2mu\sigma) = R + 2mu\sigma$, every node in S_1 that is at most a distance of $R + 2mu\sigma$ from p at the beginning of phase $\pi + 1$ knows p. Hence, the Theorem is proven. □

4.9 Discussion

In this section, we address the issue of obtaining initial neighbor knowledge, and discuss how to relax the assumption that every node knows its entire future trajectory. We also discuss practical values for the parameters.

Up till now we have focused on *maintaining* neighbor knowledge as nodes move in and out of each other's broadcast range. If nodes possess some amount of neighbor knowledge initially, then our solution allows nodes to keep this knowledge up-to-date. Specifically, we have assumed that when nodes wake up initially, they already possess

knowledge about other nodes that are within a Manhattan distance of $R + 2mu\sigma$. In this section we address a special case of this problem of gaining initial knowledge. We use the gossiping (all-to-all communication) algorithm in [30] as a black box.

For simplicity, we consider the case of the one-dimensional line. We assume that nodes start up at the same time and remain within their original segment for some period of time which we call the initialization phase. We also assume that there is an upper bound v on the number of nodes present within one segment during the initialization phase and this upper bound is known to all nodes. This is because the algorithm given in [30] requires that all nodes know the linear upper bound on the number of nodes participating in the algorithm. Note that v may be much smaller than the total number of nodes in the entire network, which is n ($v << n$). Note also that the nodes within one segment form a clique, since they are all within each other's broadcast range.

The initialization phase is divided into two parts. The first part is further divided into a total of m periods. During period i, we run an initialization protocol in segments of color i, while nodes in other segments listen. This initialization protocol is based on the algorithm for gossip in undirected graphs with unknown topology presented in [30]. This algorithm requires $O(v \log^2 v \log^2 n)$ time slots. Hence, each of the m periods in the first part of the initialization phase has $O(v \log^2 v \log^2 n)$ time slots. This algorithm performs gossiping in graphs with arbitrary topologies. Therefore, it may be used for a clique, which is the topology formed by nodes in one segment. Nodes also do not require collision detection capabilities or any previous knowledge of the neighborhood in order to run this algorithm. Furthermore, it is assumed in [30] that node ids come from a domain of size n and the actual number of nodes participating in the algorithm may be smaller than n. This assumption fits in with our model since there can be at most v nodes in one segment participating

in one period and $v << n$.

In period i of part one of the initialization phase, nodes in segments of color i run this gossiping algorithm from [30] and collect each other's ids and trajectories. Note that while the nodes of a certain segment (say s) run the algorithm, nodes in neighboring segments which are contained within distance $R - G$ of s successfully receive the transmissions by the nodes in segment s. Hence, after m periods, all nodes have received the ids and trajectories of nodes in segments contained within distance $R - G$ of their own segment. However, we require knowledge about nodes within a distance of $R + 2mu\sigma$. This can be easily achieved in the second part of the initialization phase. This part of the initialization phase is divided into two cycles. Each cycle consists of m time slots. During time slot i of both cycles, a leader node from each segment of color i simply transmits its entire id and trajectory list. The leader node is the node with the smallest id in the segment.

In order to see why this achieves the desired initial knowledge consider the following. The exact number of segments entirely contained within distance $R - G$ of a segment is given by $\lfloor (R - G)/G \rfloor$. Consider a segment s as shown in Figure 4.13. After part one all nodes in s know the ids and trajectories of all other nodes in segments within distance $(\lfloor (R - G)/G \rfloor)G$. Consider an interval I_{leader} of length $L + 2G$ that is located on the right side of s (analogous arguments hold for the left side of s) where the distance between the left end of s and the right end of I_{leader} is R. Due to the density assumption, during the initialization phase it is guaranteed that there is at least one leader node present in I_{leader}. In the worst case this leader node may remain at the left end of I_{leader} for the entire duration of the initialization phase. When this leader node broadcasts in its time slot during cycle one of part two, it will broadcast the ids and trajectories of all nodes within distance $(\lfloor (R - G)/G \rfloor)G$ of its segment, which it has collected during part one. All nodes in

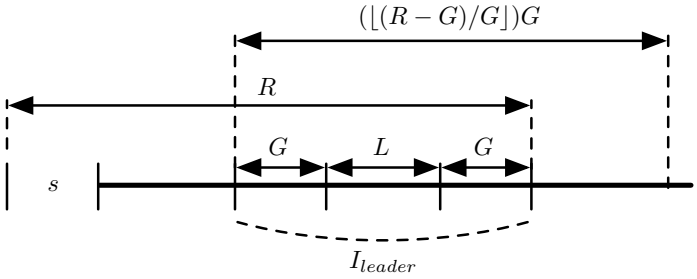


Figure 4.13: Part two of the initialization phase.

segment s will receive the leader node's broadcast successfully since the leader node is within distance R of all nodes in s. Therefore, by the end of cycle one of part two all nodes in s will know the ids and trajectories of all other nodes within distance $D_{cycle1} = R - G - (L + 2G) + (\lfloor (R-G)/G \rfloor)G$.

Now, in cycle two the same leader node (which stays in its original segment during the entire initialization phase) transmits the ids and trajectories of nodes within D_{cycle1} of itself, which it has learnt during cycle one. Therefore, by the end of cycle two of part two all nodes in s will know the ids and trajectories of all other nodes within distance $R - L - 3G + D_{cycle1}$. Assume that $L \geq G$ as in Section 4.8. Then we have the following:

$$
\begin{aligned}
& R - L - 3G + D_{cycle1} \\
= \; & R - L - 3G + (R - L - 3G + (\lfloor (R-G)/G \rfloor)G) \\
\geq \; & 2R - 2L - 6G + ((R-G)/G - 1)G \\
= \; & 3R - 2L - 8G \\
\geq \; & R - 2L - 8G + 2(2L + 3G + 8mu\sigma) && \text{(by Constraint 4.3)} \\
> \; & R + 2mu\sigma. && \text{(since } L \geq G\text{)}
\end{aligned}
$$

Hence, by the end of part two of the initialization phase all nodes in s will know the ids and trajectories of all other nodes within distance $R + 2mu\sigma$.

The challenging part of gaining initial knowledge is that the total number of unique ids present in the entire network may be very large, and only a few of these ids belong to nodes in one segment (recall that $v << n$). Learning about which of these ids belong to neighbors deterministically, without collision detection, may take a long time. However, the above protocol is deterministic and efficient, since it is sub-quadratic in the maximum number of nodes in one segment and only polylogarithmic in the total number of unique ids that may be present in the entire network.

Another important assumption that we have made is that nodes initially know their entire future trajectory. Instead of having such a strong assumption, in each round, the time slots that are not being used by our solution may be used for an intra-segment communication protocol, so that nodes may transmit their trajectory information to the current leader of the segment periodically. Hence, it may be sufficient for nodes to know their trajectories for only a short interval of time in the future. Such an intra-segment communication protocol may require an upper bound on the number of nodes that can occupy a segment at a particular instant of time. This is because only a limited number of time slots will be available in each round.

It is not hard to find practical values for the parameters that satisfy all constraints: We let R and R' to be 250 meters and 550 meters, respectively, which are typical values for IEEE 802.11 ([29, 56]). Let one time slot duration be 600 microseconds which is slightly greater than the GSM (Global System for Mobile Communications) time slot duration 577 microseconds [1]. Also, let $G = 40$ meters, $m = 30$ colors, and $u = 6$ milliseconds which corresponds to 10 time slots. Then, we can allow the upper bound on the node speed to be $\sigma < 108$ kilometers/hour and the density parameter to be $L = 40$ meters.

5. NEIGHBOR DETECTION IN WIRELESS NETWORKS WITH ARBITRARY MOTION

In this section, we provide a solution for each mobile node to keep track of its dynamically changing set of neighbors with which it can perform reliable point-to-point communication. Our solution tolerates arbitrary motion of the nodes as long as there is an upper bound on the speed, allows unsynchronized clocks subject to a bounded drift rate, and does not require any knowledge about the nodes' future locations. In addition, we use the abstract MAC layer [45] to provide reliable communication between neighboring nodes. The abstract MAC layer (AML) takes care of lower layer contention management and provides reliable broadcast with bounded message delay. This means that by using the AML, we can focus on the algorithmic aspects of the services provided on top of the AML; not worrying about scheduling wireless transmissions. We use the reliable broadcast property of the abstract MAC layer to provide reliable point-to-point communication where for each message there exists a specific destination.

We introduce a neighbor detection algorithm that utilizes periodic hello messages considering bounded clock drift. In our algorithm, two nodes do *not* become neighbors simply because they are within each other's communication radius. Instead, a node p_i considers some other node p_j as its neighbor when it is ensured that an application message sent from p_i will be received by p_j. The condition that two nodes become neighbors depends on the message delay bounds (provided by the AML), communication radius, hello period, and the maximum speed of a node. The key strategy that our algorithm uses is the following: when nodes p_i and p_j are close enough and p_i receives p_j's hello message, then p_i sets a deadline of being neighbors

with p_j; if p_i and p_j remain close enough to each other as time goes on, then this deadline is extended. More specifically, when nodes p_i and p_j are close enough and p_i considers node p_j as its neighbor by receiving a hello message generated by p_j, then the deadline of p_j being p_i's neighbor is set such that the next hello message generated by p_j is received by p_i before the deadline is reached. This is to allow p_i to continue to consider p_j as its neighbor by extending the deadline if p_j is still close enough to p_i. The algorithm also provides a way for p_j to consider p_i as its neighbor by letting p_j to refer to the deadline already calculated by p_i: If p_i was already considering p_j as its neighbor, then p_j may receive a hello message from p_i, which contains the amount of time left to reach p_i's deadline of considering p_j as its neighbor. At this point, node p_j can estimate p_i's deadline and then calculate its deadline of considering p_i as its neighbor based on this estimation.

5.1 Challenges

The main challenges in obtaining the above mentioned deadlines are as follows:

- Since we use periodic hello messages, hello messages should be generated and broadcast (sent to the AML) at the right time but, at the same time, we have to handshake with the AML.
- Since hello messages are control messages that must be generated and broadcast (sent to the AML) at the right time, they can interfere with regular application message broadcasts (application message broadcasts can be blocked by hello message broadcasts).
- Since we consider bounded clock drift, different nodes may have different hello periods when measured in real time.

Our algorithm overcomes the above challenges in providing neighbor detection with reliable point-to-point communication.

5.2 Related Work.

Cornejo et al. [19] consider the use of abstract MAC layer for neighbor detection. In their approach, the geographic space is divided into regions. Also, each node knows its trajectory information up to the near future such that it can correctly join or leave a region by notifying other neighboring nodes that it is *planning* to join or leave the region. Similar to [19], our approach utilize the abstract MAC layer, however, we do not require region information nor trajectory information. Instead, we require periodic hello messages and each node knowing its correct location.

An ALOHA-like neighbor discovery algorithm is presented in [72]. The authors further extend this algorithm by relaxing clock requirements, allowing different node wake up times, not knowing the total number of nodes in the system, and adding collision detection capability. The analysis of their algorithm is based on all nodes forming a complete graph. In our approach, there is absolutely no restriction on the network topology.

Liu [48] considers neighbor discovery in an environment where malicious nodes can deceive benign nodes. The solution is based on nodes realizing tentative neighbors and then using a neighbor validation function to see if they are truly neighbors. Even though node deployment is not restricted to a certain network topology, the paper considers the case of nodes being static after deployment. In our approach, nodes are mobile and there is no restriction on the movement of nodes. However, we do not consider malicious failures.

Neighbor discovery in multi-channel radio networks is discussed in [42] and [43]. Depending on each node's id (to determine the assigned synchronous time slot) and channel availability, a schedule for wireless transmission is determined. In our approach, we delegate such scheduling to the abstract MAC layer and we do not

require the clocks to be synchronized.

5.3 System Model

We consider a collection of mobile nodes sending and receiving messages among each other by means of wireless broadcasts. We assume that each node has a unique id, knows its own fixed communication radius (the communication radius can be different for different nodes), and knows the maximum speed (denoted as σ) that any node can move in the system.

Each node's clock may not run at the same rate, however, clock drift is upper bounded by ρ ($0 \leq \rho < 1$) with respect to real time: if Δ real time elapses, then the amount of clock time that elapses is at most $(1 + \rho)\Delta$ and at least $(1 - \rho)\Delta$. We further assume that the value of ρ is known to all nodes in the system. We do *not* assume that the clocks are synchronized.

We further assume that nodes are correct, do not crash, and have equipment that lets them know their correct location at any time. We also consider that the local processing time at each node is 0.

Each node is organized with three layers (see Figure 5.1). The three layers are described next.

The abstract MAC layer (AML) ensures reliable message delivery despite contention. It also provides maximum message delay bounds for upper layers to utilize. We assume that message delay bounds, F^+_{rcv} and F^+_{ack} from the abstract MAC layer, are known to all nodes in the system where F^+_{rcv} indicates the maximum message delay for a node to receive a message (by $rcv(m)$ in Figure 5.1) from another node within communication radius and F^+_{ack} indicates the maximum delay of a node receiving an acknowledgement (by $ack(m)$ in Figure 5.1) for a message that it transmitted. Typically, $F^+_{ack} \geq F^+_{rcv}$ (this is also assumed throughout Section 5). In order to utilize

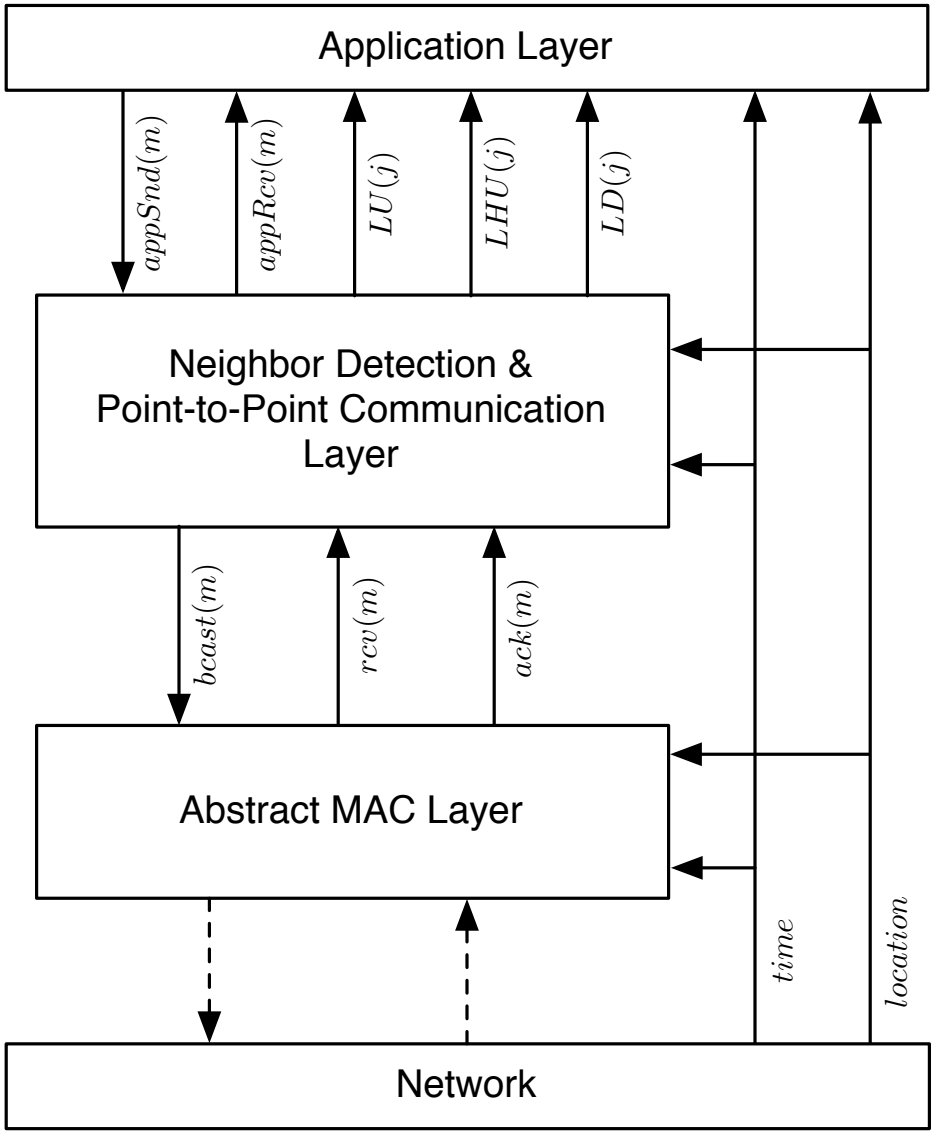


Figure 5.1: Layers of a node.

message delay bounds, upper layers must follow the *well-formedness* condition of the AML: a message should not be sent to the AML ($bcast(m)$ in Figure 5.1) unless an acknowledgement of the previous message ($ack(m)$ in Figure 5.1) has been returned by the AML.

The neighbor detection & point-to-point communication layer (ND/P2P) is responsible for detecting neighboring nodes and notifying applications about the the current status (link up, link down, etc.) of neighboring nodes. As seen in Figure 5.1, notifications are through $LU(j)$ (link up), $LD(j)$ (link down), and $LHU(j)$ (link half-up). $LU(j)$ occurring at node p_i indicates that a communication link has been established between p_i and p_j such that applications of p_i can send/receive messages to/from applications of p_j. $LHU(j)$ occurring at node p_i indicates that a communication link has been established between p_i and p_j such that applications of p_i can receive messages from applications of p_j but cannot send messages to p_j. And finally, $LD(j)$ occurs at node p_i indicates that applications of p_i cannot send/receive messages to/from applications of p_j. We refer to $LU(j)$, $LHU(j)$, and $LD(j)$ as link state events. When $LU(j)$ (resp. $LHU(j)$; $LD(j)$) is the most recent link state event, we say that the link state is link up (resp. link half-up; link down). Furthermore, the ND/P2P layer is responsible for sending application messages to the AML. In order to conform with the well-formedness condition of the AML, the ND/P2P layer includes a k-bounded send buffer where each message sent by the application layer ($appSnd(m)$ in Figure 5.1) is stored and sent to the AML one-by-one. Upon receiving an application message (generated by a different node) from the AML (by $rcv(m)$ in Figure 5.1), the ND/P2P layer is also responsible for sending the received message to the application layer (by $appRcv(m)$ in Figure 5.1).

The application layer models the set of applications that utilize the notifications provided by the ND/P2P layer ($LU(j)$, $LHU(j)$, and $LD(j)$ in Figure 5.1) and also

sends/receives messages to/from the ND/P2P layer ($appSnd(m)$ and $appRcv(m)$ in Figure 5.1). We assume that the set of applications in the application layer at node p_i do not send messages to node p_j if the most recent link state event with respect to p_j is not $LU(j)$. For simplicity, we also assume that the set of applications in the application layer send messages to the ND/P2P layer at a rate that does not overflow the k-bounded send buffer. This assumption can be implemented by a hand-shaking mechanism between the applications and the ND/P2P layer.

An execution γ is a sequence of events happening in the system where the events are either application events ($appSnd(m)$ and $appRcv(m)$ in Figure 5.1), link state events ($LU(j)$, $LHU(j)$, and $LD(j)$ in Figure 5.1), abstract MAC layer events ($bcast(m)$, $rcv(m)$, and $ack(m)$ in Figure 5.1), or internal events (timers expiring at each nodes). Each event in γ is associated with the real time at which it occurs. Additional constraints on the execution, that are related to the problem of neighbor detection, will be given in Section 5.4.

5.4 The Problem Specification

Let $LS_i^t(j)$ refer to the most recent link state event occurring at node p_i with respect to node p_j at or before time t. $LS_i^t(j)$ can be either LU (link up), LHU (link half up), or LD (link down).

For any execution γ of the system, we require the following constraints on γ:

- (*Usability*) An application send ($appSnd(m)$ in Figure 5.1) should not occur unless the current link state is LU and an application receive ($appRcv(m)$ in Figure 5.1) should not occur if the current link state is LD.
- (*Validity*) For all real times t, if $LS_i^t(j) = LU$, then p_i and p_j are within each other's communication radius at time t.
- (*Well-formedness*) For all real times t and t' where $t < t'$, suppose $LS_i^t(j) =$

LU and $LS_i^{t'}(j) = LD$. Then there exists a real time $t'' \in (t, t')$ such that $LS_i^{t''}(j) = LHU$; The link state does not change directly from LU to LD.

- (*Progress*) There exists positive reals x_p, α'_p, and α''_p where $\alpha'_p \leq \alpha''_p$ such that, for every real $\alpha_p \geq 0$, if the distance between two nodes p_i and p_j is at most x_p for a time duration of $[t, t + \alpha_p]$ (in real time), then both $LS_i^{t'}(j) = LU$ and $LS_j^{t'}(i) = LU$ for all times $t' \in [t + \alpha'_p, t + \alpha_p + \alpha''_p]$ (in real time); Roughly speaking, this property says that if two nodes p_i and p_j stays close enough with each other, then there is a time in the near future where both p_i and p_j will be link up with each other.
- (*Reliable Delivery*) Each application send has a subsequent matching application receive and each application receive has exactly one preceding application send.

5.5 The Neighbor Detection Algorithm

We consider an algorithm where periodic hello messages are utilized in establishing communication links. The algorithm uses two parameters, h and b (in addition to ρ, σ, F^+_{rcv}, and F^+_{ack}), which we explain next. We assume that the value of the hello period duration h in clock time is known to all nodes in the system. We select the hello period h to satisfy $h \geq (b+2)F^+_{ack} \cdot \frac{1+\rho}{1-\rho}$ where b is a parameter which indicates the maximum number of application message broadcasts to the AML that we want to allow during a hello period considering each broadcast taking F^+_{ack} amount of real time. Each hello message of node p_i consists of the following five fields: 1) *id* : node id of p_i; 2) *location* : p_i's location when the hello message is generated; 3) *radius* : communication radius of p_i; 4) *neighbors* : p_i's current neighbor set (all nodes p_j where $LS_i(j)$ is either LU or LHU); 5) *remaining* : an array where $remaining[j]$ indicates the amount of local clock time left until reaching the time

that the applications of p_i should stop sending messages to node p_j.

To allow hello messages to be sent to the AML every h local time units while satisfying the well-formedness condition of the AML, we disallow application messages to be sent to the AML in the local time duration of $[t - (1+\rho)F^+_{ack}, t]$ where t is each local time the hello message is scheduled to be sent to the AML (note that the amount of local time $(1+\rho)F^+_{ack}$ in real time is always greater than or equal to the amount of real time F^+_{ack}). We call this time duration the *pre-hello* interval. Note that since we are assuming clock drift, nodes may generate hello messages at different times (in real time).

For our algorithm, we assume that $kF^+_{ack} \geq h/(1+\rho)$. We also assume the following functions for our algorithm:nd:

- Loc(): returns the current location; provided by the GPS.
- LC() : returns the current local clock time.
- closeEnough($loc_1, rad_1, loc_2, rad_2$) : returns true *iff* the following formula is satisfied :

  $dist(loc_1, loc_2) + 2\sigma((h/(1-\rho) + F^+_{rcv} + 2\lceil \frac{kF^+_{ack}}{h/(1+\rho)} \rceil F^+_{ack}) \cdot \frac{1+\rho}{1-\rho} + kF^+_{ack} + F^+_{rcv}) \leq \min(rad_1, rad_2)$ where function $dist(\phi_1, \phi_2)$ returns the Euclidean distance between location ϕ_1 and ϕ_2.

  - 2σ corresponds to the maximum speed in which two nodes moving away from each other.
  - In our algorithm, closeEnough() is called at some node p_i only when a hello message is received from some other node p_j. The parameters passed to closeEnough() are p_i's current location, p_i's communication radius, p_j's location when the hello message was generated, and p_j's communication radius. If closeEnough() returns true at p_i, then p_i sets a deadline of which p_i should stop sending application messages to p_j. The deadline

is determined under the condition that before the deadline is reached, the next hello message from p_j should be received by p_i. In the above formula, $(h/(1-\rho) + F^+_{rcv}) \cdot \frac{1+\rho}{1-\rho}$ accounts for the amount of time (in real time) in reaching this deadline (in every h local time units a hello message is generated and it takes at most F^+_{rcv} (real) time to receive the hello message).

- An additional F^+_{rcv} accounts for the time the hello message of p_j was in transit.
- $kF^+_{ack} + 2\lceil \frac{kF^+_{ack}}{h/(1+\rho)} \rceil F^+_{ack} \cdot \frac{1+\rho}{1-\rho}$ accounts for the total (real) time in emptying the k-bounded send buffer if the buffer is full. kF^+_{ack} corresponds to the maximum overhead in emptying the k-bounded send buffer when application message broadcast has is no interference with control (hello) message broadcast. However, since, in our case, hello messages are broadcast every h local time units, hello message broadcasts may interfere with application message broadcasts. $2\lceil \frac{kF^+_{ack}}{h/(1+\rho)} \rceil F^+_{ack} \cdot \frac{1+\rho}{1-\rho}$ accounts for the total time of hello messages interfering with the k consecutive messages in the send buffer. This is because $\lceil \frac{kF^+_{ack}}{h/(1+\rho)} \rceil$ corresponds to the maximum number of hello message broadcasts during kF^+_{ack} amount of (real) time and for each hello message broadcast there is a pre-hello interval and a F^+_{ack} time for receiving an ack for the transmitted hello message.
- Now, suppose closeEnough() returned true at p_i by receiving a hello message generated by p_j. We can deduce from the above that p_j will receive p_i's application messages that were sent before reaching the deadline since even after p_i sends an application message just before the deadline, p_i and p_j will be within each other's communication radius for a time duration of emptying the k-bounded send buffer even though p_i and p_j are moving

away from each other in maximum speed.

Furthermore, we define the following macros for our algorithm :

- $Neighbors$: $\{j \mid status[j] = LU$ or $LHU\}$
- SU_1 : $rmn[i] \cdot \frac{1-\rho}{1+\rho} - (1+\rho)F^+_{rcv}$
- RU_1 : If $rmn[i] \geq 0$, then $rmn[i] \cdot \frac{1+\rho}{1-\rho} + (1+\rho)(kF^+_{ack} + 2\lceil \frac{kF^+_{ack}}{h/(1+\rho)} \rceil F^+_{ack} \cdot \frac{1+\rho}{1-\rho})$.
  Otherwise, $rmn[i] \cdot \frac{1-\rho}{1+\rho} + (1+\rho)(kF^+_{ack} + 2\lceil \frac{kF^+_{ack}}{h/(1+\rho)} \rceil F^+_{ack} \cdot \frac{1+\rho}{1-\rho})$.
- SU_2 : $h \cdot \frac{1+\rho}{1-\rho} + (1+\rho)F^+_{rcv}$
- RU_2 : $h \cdot \frac{1+\rho}{1-\rho} + (1+\rho)(F^+_{rcv} + kF^+_{ack} + 2\lceil \frac{kF^+_{ack}}{h/(1+\rho)} \rceil F^+_{ack} \cdot \frac{1+\rho}{1-\rho})$

Formulas SU_1, RU_1, SU_2, and RU_2 are used in the algorithm when a hello message is received. We first explain SU_2 and RU_2. Suppose node p_i receives a hello message from p_j and closeEnough() returned true at p_i at p_i's local time t. Then, the deadline for p_i to stop sending application messages to p_j is set as $t + SU_2$ (called the *sending deadline*) and the deadline for p_i to even stop receiving application messages from p_j is set as $t + RU_2$ (called the *receiving deadline*). SU_2 and RU_2 are both based on the formula that makes closeEnough() to return true. However, they refer to local clock time rather than real time. First, note that SU_2 corresponds to the amount of (local) time of p_i receiving p_j's next hello message even if p_i's clock runs at the fastest rate and p_j's clock runs at the slowest rate. Next, note that $SU_2/(1-\rho) = (h/(1-\rho) + F^+_{rcv}) \cdot \frac{1+\rho}{1-\rho}$. This means that even if p_i's clock runs at the slowest rate, the amount of local time SU_2 in real time will not exceed the amount of real time $(h/(1-\rho) + F^+_{rcv}) \cdot \frac{1+\rho}{1-\rho}$. So, since closeEnough() returned true, we can deduce that p_j will receive p_i's application messages that were sent before reaching $t + SU_2$. Formula RU_2 simply considers the amount of (local) time in emptying the k-bounded send buffer after the sending deadline $t + SU_2$.

We now explain SU_1 and RU_1. Suppose node p_i receives a hello message $hello_j$ from p_j at p_i's local time t where $i \in hello_j.neighbors$ (p_i was considered as p_j's

neighbor at the time when $hello_j$ was generated). Then, formulas SU_1 and RU_1 are used in setting the sending and receiving deadlines at p_i (as $t + SU_1$ and $t + RU_1$, respectively) referring to $hello_j.remaining[i]$ (=$rmn[i]$) which was set based on p_j's local clock. Setting p_i's deadlines in this way is to allow p_i to consider p_j as its neighbor even though closeEnough() did not return true (p_i gets a chance to send application messages to p_j). Since p_i's clock and p_j's clock may run at different rates, p_i can only *estimate* the deadlines set at p_j. In estimating p_j's sending deadline, it is important that p_i does *not* set its sending deadline to be greater than p_j's sending deadline when both deadlines are transformed into real time. Formula SU_1 guarantees this since 1) the amount of p_i's local time ($hello_j.remaining[i]) \cdot \frac{1-\rho}{1+\rho}$ in real time does not exceed the amount of p_j's local time $hello_j.remaining[i]$ in real time even if p_i's clock runs at the slowest rate and p_j's clock runs at the fastest rate, and 2) it also considers the maximum (local) time $hello_j$ might have been in transit ($(1+\rho)F^+_{rcv}$). In determining p_i's receiving deadline, it is important that p_i's receiving deadline is set such that p_i receives all application messages from p_j that was sent before reaching p_j's sending deadline. Formula RU_1 guarantees this since, in calculating the p_i's receive deadline, p_i first considers the greatest possible value of the sending deadline in local time referring to $hello_j.remaining[i]$ (the situation where $hello_j$ is instantaneously received by p_i and p_i's clock runs at the fastest rate (resp. slowest rate) while p_j's clock runs at the slowest rate (resp. fastest rate) considering $hello_j.remaining[i] \geq 0$ (resp. $hello_j.remaining[i] < 0$)) and then adds the the amount of (local) time in emptying the k-bounded send buffer.

We now describe the neighbor detection algorithm (the ND/P2P layer of node p_i). The pseudocode is given in Figures 5.2 and 5.3.

There are seven main variables used in the algorithm: 1) R stores the communication radius of node p_i, 2) $sendUntil$ is an array where $sendUntil[j]$ indicates the

⟨**Variables and Initialization**⟩
1: $R \leftarrow$ communication radius of node p_i;
2: *sendUntil*; // array[1..n]; initialized to 0
3: *receiveUntil*; // array[1..n]; initialized to 0
4: *remaining*; // array[1..n]; initialized to 0
5: *sendQueue*; // the k-bounded send queue; initialized to $\emptyset$
6: $recvBuff$; // array[1..n] of queues; initialized to $\emptyset$
7: *state*; // array[1..n]; initialized to LD

8: ⟨**When** LC() is a multiple of h⟩
9: $\forall j \in Neighbors : remaining[j] \leftarrow sendUntil[j] - \text{LC}()$;
// send a hello msg to AML
10: broadcast $hello(i, \text{Loc}(), R, Neighbors, remaining)$;

11: ⟨**When** $hello(j, loc, rad, nbrs, rmn)$ is received⟩
12: **if** $i \in nbrs$ **then**
13: updateDeadlines(j, LC() + SU_1, LC() + RU_1);
14: **if** closeEnough(Loc(),R,loc,rad) **then**
15: updateDeadlines(j, LC() + SU_2, LC() + RU_2);

16: ⟨**When** LC() = $sendUntil[j]$⟩
17: $state[j] \leftarrow LHU$;
18: enable $LHU(j)$; // an event that goes to the application layer

19: ⟨**When** LC() = $recvUntil[j]$⟩
20: $state[j] \leftarrow LD$;
21: enable $LD(j)$; // an event that goes to the application layer

22: ⟨**When** app msg m is received from application layer⟩
23: enqueue(*sendQueue*,m);

Figure 5.2: ND/P2P layer; code for node p_i (part 1 of 2).

```
24: ⟨When (sendQueue ≠ ∅) ∧ (LC() is not in a pre-hello interval) ∧
      (p_i is not waiting for an ack of a previously broadcast msg from the AML)⟩
25: broadcast dequeue(sendQueue); // send to AML

26: ⟨When app msg m is received from p_j⟩
27: if m is destined for p_i then
28:    if state[j] ≠ LD then
29:       deliver m to application layer;
30:    else
31:       enqueue(recvBuff[j],m); // store msgs that arrived "early"

32: procedure updateDeadlines(j, s, r)
33:    if sendUntil[j] < s then
34:       sendUntil[j] ← s;
35:       if state[j] ≠ LU then
36:          state[j] ← LU;
37:          deliverPendingAppMsgs(j); // deliver msgs that arrived "early"
38:          enable LU(j); // an event that goes to the application layer
39:    if recvUntil[j] < r then
40:       recvUntil[j] ← r;
41:       if state[j] = LD then
42:          state[j] ← LHU;
43:          deliverPendingAppMsgs(j); // deliver msgs that arrived "early"
44:          enable LHU(j); // an event that goes to the application layer

45: procedure deliverPendingAppMsgs(j)
46:    deliver each app msg in recvBuff[j] to application layer;
```

Figure 5.3: ND/P2P layer; code for node p_i (part 2 of 2).

local time that the applications of p_i should stop sending messages to node p_j (the sending deadline), 3) *receiveUntil* is an array where $receiveUntil[j]$ indicates the local time that the applications of p_i should stop receiving messages from node p_j (the receiving deadline), 4) *remaining* is an array that is only temporarily used when a hello message is constructed where $remaining[j]$ indicates the local time remaining until applications of p_i should stop sending messages to node p_j, 5) *sendQueue* corresponds to the k-bounded send queue, 6) $recvBuff$ is an array of queues where $recvBuff[j]$ stores the messages received by p_j when the link state is LD, and finally 7) *state* is an array where $state[j]$ indicates the current link state with respect to p_j.

In every h local time units, a hello message is broadcasted (sent to the AML). When node p_i receives a hello message from node p_j, p_i first checks if the hello message contains the information regarding p_i being p_j's neighbor. If so, then p_i attempts to update $sendUntil[j]$ and $recvUntil[j]$ by calling updateDeadlines(). The parameter values passed to updateDeadlines() are *estimates* of $sendUntil[i]$ and $recvUntil[i]$ at p_j (LC()+SU_1 and LC()+RU_1). Procedure updateDeadlines() only allows $sendUntil[j]$ and $recvUntil[j]$ to be updated when the parameters passed to updateDeadlines() is greater than $sendUntil[j]$ and/or $recvUntil[j]$. If $sendUntil[j]$ gets to be updated, then the link state with respect to p_j either remains or changes to LU. And, if $recvUntil[j]$ gets to be updated, then the link state with respect to p_j either remains or changes to LU or LHU. If the link state with respect to p_j *changed* to LU or LHU, then each application message from p_j in $recvBuff[j]$ (messages that arrived "early" from p_j) will be delivered to the application layer.

If p_i receives a hello message from node p_j and a call to closeEnough() returns true (line 14 in Figure 5.2), then updateDeadlines() is called with different parameter values than the values mentioned in the previous paragraph. These parameter values (LC()+SU_2 and LC()+RU_2) guarantee that the link state will be LU (if it was not

already LU) and while the link state is LU, p_i will receive p_j's next hello message (see Lemmas 5.6.7 and 5.6.9 in Section 5.6).

If the sending deadline with respect to node p_j is reached (LC()=$sendUntil[j]$) at p_i, then p_i changes its link state with respect to p_j to LHU. This stops applications from sending messages to p_j while still allows them to receive messages from p_j. And, if the receiving deadline with respect to node p_j is reached (LC()=$recvUntil[j]$) at p_i, then p_i changes its link state with respect to p_j to LD. This makes the applications of p_i to stop sending and receiving messages regarding p_j.

When an application message is received by the application layer, then the message is simply enqueued in the k-bounded send buffer *sendQueue*.

In order to broadcast a message in the *sendQueue* (sending it to AML), the following three conditions must be satisfied: 1) *sendQueue* must not be empty, 2) the current local time (LC()) must not be in the pre-hello interval, 3) the ND/P2P layer must not be waiting for an ack of a previously broadcast message from the AML (this is to conform with the well-formedness condition of the AML).

When an application message is received from node p_j at p_i, then depending on the current link state, the received message is either directly delivered to the application or stored in $recvBuff[j]$: if the current link state is LU or LHU, then deliver the received message to the application layer; otherwise, store it in $recvBuff[j]$ for later delivery when the link becomes either LU or LHU. This is to prevent applications from receiving messages while the current link state is LD.

5.6 Proof of Correctness

An overview of the proofs is as follows. Our objective is to prove that the five properties mentioned in Section 5.4 are satisfied. The usability property (Lemma 5.6.1) is trivially proven by inspecting the code. In proving the well-formedness

property (Theorem 5.6.6), we show that when a node is link up with some other node p_j, the deadline for sending messages to p_j is reached (causing the link state to change to LHU) prior to changing the link state with respect to p_j to LD by utilizing the fact that the link is up with respect to p_j at time t if and only if the deadline for sending messages to p_j is some time after t (Lemma 5.6.5).

The proof of the validity property (Theorem 5.6.8) is based on how the deadline for sending messages to a certain node is set (again using Lemma 5.6.5). There are two ways in setting the deadline: executing line 15 or line 13 of Figure 5.2. In the case of executing line 15, we show the desired result by using the fact that the function closeEnough() returning true has the meaning of two nodes being within each other's communication radius for a certain time duration (Lemma 5.6.7). In the case of executing line 13, we reason about how formula SU_1 gives us the desired result.

The proof of the progress property (Theorem 5.6.10) is based on the fact that when two nodes are close enough, they will receive each other's hello message (Lemma 5.6.9) even though they move away from each other and by receiving each other's hello message, function closeEnough() will return true which will cause the link state to be LU for both nodes.

In proving the reliable delivery property (Theorem 5.6.11), we first use the *no duplication* property of the AML (no two receive events are caused by the same broadcast event) to show that no two application receives are caused by the same application send. Then, we show that application messages sent to p_j by p_i while the link is up are guaranteed to be received by the ND/P2P layer of p_j. Finally it is shown that the application message at the ND/P2P layer of p_j will be delivered to the application layer of p_j.

Throughout the analysis, we denote the value of variable var of node p_i at time

t as var_i^t. Fix γ to be an arbitrary execution of the algorithm in Figures 5.2 and 5.3.

By the assumption that an application send occurs only when the current link state is up and by the fact that an application receive does not occur while the link state is LD (line 31 of the pseudocode), we immediately get the following lemma:

Lemma 5.6.1. *γ satisfies the usability property.*

The next lemma shows that the values of both $sendUntil$ and $recvUntil$ does not decrease over time.

Lemma 5.6.2. *For all nodes p_i and p_j and for all times t and t' where $t \leq t'$, $sendUntil_i^t[j] \leq sendUntil_i^{t'}[j]$ and $recvUntil_i^t[j] \leq recvUntil_i^{t'}[j]$.*

Proof. The proof is straightforward from the fact that (1) the only way to update $sendUntil_i[j]$ and/or $recvUntil_i[j]$ is by calling updateDeadlines(), and (2) updateDeadline() only allows updates to $sendUntil_i[j]$ and/or $recvUntil_i[j]$ when a greater parameter value is passed to it (lines 33 and 39). □

The following lemma and corollary shows that at any given time, the value of $sendUntil[j]$ is at most the value of $recvUntil[j]$.

Lemma 5.6.3. *Suppose, at time t_1, updateDeadlines(j, _, _) is executed for the first time at node p_i by receiving a hello message from node p_j. Then, $sendUntil_i^t[j] < recvUntil_i^t[j]$ for all times $t \geq t_1$.*

Proof. When updateDeadlines(j, s, r) is called at lines 13 or 15 at some time t', we have $0 < s < r$ since $0 < t' + SU_1 < t' + RU_1$ and $0 < t' + SU_2 < t' + RU_2$. Since the only way to update $sendUntil_i[j]$ and $recvUntil_i[j]$ is by calling updateDeadlines(j, _, _) and since $sendUntil_i[j]$ and $recvUntil_i[j]$ is updated only when their current value is less than s and r, respectively, it is straightforward that $sendUntil_i^t[j] < recvUntil_i^t[j]$ for all times $t \geq t_1$. □

Corollary 5.6.4. *For all nodes p_i and p_j and for all times t, $sendUntil_i^t[j] \leq recvUntil_i^t[j]$.*

Proof. The proof is immediate from Lemma 5.6.3 and the fact that for all nodes, the elements of $sendUntil$ and $recvUntil$ are initialized to 0. □

The following lemma is the key lemma in proving the well-formedness and validity property which says that the link state is link up with respect to some node p_j at a given time t if and only if the value of $sendUntil[j]$ is greater than t.

Lemma 5.6.5. *For all nodes p_i and p_j and for all times t, $LS_i^t(j) = LU$ if and only if $sendUntil_i^t[j] > t$ (in p_i's local time).*

Proof. ($\Rightarrow$) Suppose, in contradiction, that $sendUntil_i^t[j] \leq t$. This implies that there exists a local time $t'(\leq t)$ such that $t' = sendUntil_i^t[j]$. If $t' = 0$, then we have $LS_i^{t'}(j) = LD$ by the initialization code. If $t' \neq 0$, then we have $LS_i^{t'}(j) = LHU$ at local time t' by line 17. Hence, at local time t', we have a link state that is not LU. Now, since $LS_i^t(j) = LU$, there exists a local time during $[t', t]$ where line 36 is executed. In order to execute 36, $sendUntil_i[j]$ must have been modified to a greater value than t' during $[t', t]$. Since $sendUntil_i[j]$ is nondecreasing over time by Lemma 5.6.2, it must be that $sendUntil_i^t[j] > t'$, a conradiction.
($\Leftarrow$) Suppose, in contradiction, that $LS_i^t(j) \neq LU$. We distinguish two cases:
–(*Case* 1; $LS_i^t(j) = LD$): In this case, we have either $sendUntil_i^t[j] = 0$ by the initialization code or $recvUntil_i^t[j] \leq t$ by the fact that the only case the link state becomes LD is when LC()= $recvUntil_i[j]$ (line 20). If $sendUntil_i^t[j] = 0$, then we directly get a contradiction since $sendUntil_i^t[j] > t \geq 0$. If $recvUntil_i^t[j] \leq t$, then by Corollary 5.6.4, $sendUntil_i^t[j] \leq recvUntil_i^t[j] \leq t$, a contradiction.
–(*Case* 2; $LS_i^t(j) = LHU$): This case implies that there exists a time prior or equal to

time t such that the link state became LHU. Let $t'(\leq t)$ be the latest local time that the link state became LHU. By the code, either line 42 or 17 was executed at time t'. Also, since the link state remains as LHU during $[t', t]$, we have $sendUntil_i^{t'}[j] = sendUntil_i^t[j]$. If line 17 was executed at time t', then it is straightforward that $t' = sendUntil_i^{t'}[j]$. Thus, we get $sendUntil_i^t[j] = t' \leq t$, a contradiction. If line 42 was executed at time t', then the latest link state prior to time t' must have been LD by line 41. This implies that there exists a time prior to time t' such that the link state became LD. Let $t''(< t')$ be the latest local time that the link state became LD. We have $recvUntil_i^{t''}[j] = t''$ by line 20. During $[t'', t')$ updateDeadlines() must not have been called because otherwise the link state must have changed to either LU or LHU by the fact that $sendUntil_i^{t''}[j] \leq recvUntil_i^{t''}[j] = t''$ (Corollary 5.6.4) and the fact that updateDealine() is called with parameters that have a values greater than the current local time value. Moreover, the if-condition at line 33 should have been evalutated to false because otherwise the if-condition at line 41 would have evaluated to false. Hence, $sendUntil_i[j]$ was not updated during $[t'', t']$ and we get $sendUntil_i^{t''}[j] = sendUntil_i^{t'}[j] = sendUntil_i^t[j]$. Therefore, since $sendUntil_i^{t''}[j] \leq recvUntil_i^{t''}[j] = t''$, we finally get $sendUntil_i^t[j] \leq t'' < t$, a contradiction. □

The following theorem shows that the well-formedness property holds. It uses both lemmas 5.6.3 and 5.6.5 in obtaining a contradiction in the situation where the link state changes directly from LU to LD.

Theorem 5.6.6. *γ satisfies the well-formedness property.*

Proof. Suppose, for two nodes p_i and p_j, that $LS_i^t(j) = LU$ and $LS_i^{t'}(j) = LD$ where $t < t'$ in real time. Now, suppose, in contradiction, that for all real times $t'' \in (t, t')$, $LS_i^{t''}(j) \neq LHU$. Since $LS_i^{t'}(j) = LD$, we have $recvUntil_i^{t'}[j] \leq t'$ in real time. Also, since $LS_i^t(j) = LU$, we know that updateDeadlines() is called at least once. So, by

Lemma 5.6.3 and Lemma 5.6.5, we get $t < sendUntil_i^{t'}[j] < recvUntil_i^{t'}[j] \leq t'$ in real time. This implies that there exists a real time $t_1 \in (t, t')$ such that $sendUntil_i^{t_1}[j] = t_1$ in real time which results in executing line 17, a contradiction. □

The next lemma shows that the function closeEnough() returning true has the meaning of two nodes being within each other's communication radius for a certain time duration.

Lemma 5.6.7. *Suppose, at real time t, node p_i received a hello message from node p_j and closeEnough() returned true. Then, p_i and p_j will be in each other's communication radius during $[t, t+T]$ in real time where $T = (h/(1-\rho) + F_{rcv}^+ + 2\lceil \frac{kF_{ack}^+}{h/(1+\rho)} \rceil F_{ack}^+) \cdot \frac{1+\rho}{1-\rho} + kF_{ack}^+$.*

Proof. If instantaneous message delivery is assumed, the condition that needs to be satisfied for closeEnough() to return true simply guarantees that p_i and p_j remain in each other's communication radius during $[t, t + T + F_{rcv}^+]$ in real time even though p_i and p_j moves in the maximum speed ($=\sigma$) away from each other. However, in our case, the hello message generated by p_j might have took F_{rcv}^+ real time to reach p_i. Hence, we can say that p_i and p_j will be in each other's communication radius during $[t, t + T]$ in real time. □

The following theorem shows that the validity property holds. Using Lemma 5.6.5, the proof is based on how $sendUntil_i[j]$ is set at p_i.

Theorem 5.6.8. *γ satisfies the validity property.*

Proof. First, let $T = (h/(1-\rho) + F_{rcv}^+ + 2\lceil \frac{kF_{ack}^+}{h/(1+\rho)} \rceil F_{ack}^+) \cdot \frac{1+\rho}{1-\rho} + kF_{ack}^+$. Suppose $LS_i^t(j) = LU$ at local time t. By Lemma 5.6.5, we know that $LS_i^t(j) = LU$ if and only if $sendUntil_i^t[j] > t$. We distinguish two cases based on how $sendUntil_i[j]$ was set:

–(*Case* 1; By line 15 (at local time t')): In this case, the call to closeEnough() returned true. When closeEnough() returns true, it is guaranteed by Lemma 5.6.7 that p_i and p_j remain in each other's communication radius for at least T amount of time in real time from time t'. At line 15, we set $sendUntil_i[j] = t' + SU_2$. Since $0 \leq \rho < 1$, the amount of local time SU_2 in real time is always less than the amount of real time T. Hence, p_i and p_j will remain in each other's communication radius until time $sendUntil_i[j]$ in p_i's local time.

–(*Case* 2; By line 13): In this case, $sendUntil_i[j]$ was updated by $hello_j.remaining[i]$. For two nodes p_i and p_j where $LS_i^{t_1}(j) = LD$ and $LS_j^{t_1}(j) = LD$ for some real time t_1, respectively, line 15 must be executed first to execute line 13 since either $j \in Neighbors_i$ or $i \in Neighbors_j$ must first be satisfied to execute line 13. Hence, for this case, it suffices to prove that (1) when $sendUntil_i[j]$ is set by executing line 15 at p_i at time t, p_i and p_j are within each other's communication radius until time $sendUntil_i^t[j]$ and (2) assuming that $hello_j$ was generated by p_j at p_j's local time t_j and it is guaranteed that p_i and p_j remain within each other's communication radius at least until $t_j + hello_j.remaining[i]$ in p_j's local time, if $sendUntil_i[j]$ is set by executing line 13 at p_i at time t, then p_i and p_j are within each other's communication radius until time $sendUntil_i^t[j]$. Part (1) is already proven in (Case 1). We focus on part (2). By the code, $sendUntil_i[j]$ is set by the formula $t_i + hello_j.remaining[i] \cdot \frac{1-\rho}{1+\rho} - (1+\rho)F_{rcv}^+$ where t_i is p_i's local time when p_i receives $hello_j$. First note that $hello_j.remaining[i] > 0$ because otherwise $sendUntil_i[j]$ will not have been set in the first place. Since $0 \leq \rho < 1$, the amount of p_j's local time $hello_j.remaining[i]$ in real time is always greater than or equal to the amount of p_i's local time $hello_j.remaining[i] \cdot \frac{1-\rho}{1+\rho}$ in real time. In worst case, $hello_j$ might have took F_{rcv}^+ real time from when p_j generating $hello_j$ to p_i receiving $hello_j$. Again, since $0 \leq \rho < 1$, the amount of p_i's local time $(1+\rho)F_{rcv}^+$ in real time is always greater than

or equal to F^+_{rcv} in real time. Hence, we get that p_j's local time $t_j + hello_j.remaining[i]$ in real time is greater than or equal to p_i's local time $sendUntil_i^{t_i}[j]$ in real time. Therefore, p_i and p_j will remain in each other's communication radius until time $sendUntil_i^{t_i}[j]$ in p_i's local time. □

The following lemma shows how two nodes remaining in each other's communication radius for a specific time duration implies that both nodes will receive each other's hello message.

Lemma 5.6.9. *Suppose nodes p_i and p_j remain in each other's communication radius for the entire time duration of $[t, t + h/(1-\rho) + F^+_{rcv}]$ in real time. Then, p_i and p_j will receive at least one of each other's hello message during $[t, t + h/(1-\rho) + F^+_{rcv}]$ in real time.*

Proof. Since the clock drift is bounded by ρ where $0 \leq \rho < 1$ and each node generates and broadcasts a hello message every h time units with respect to its local clock, each node generates and broadcasts at least one hello message during a time duration of size $h/(1-\rho)$ in real time (the amount of time $h/(1-\rho)$ in real time corresponds to the amount of local time h for the node with the slowest clock rate). Hence, considering the message delay bound F^+_{rcv} in real time, p_i and p_j receive each other's hello message during $[t, t + h/(1-\rho) + F^+_{rcv}]$ in real time. □

The next theorem shows that the progress property holds for certain values of x_p, α'_p, and α''_p.

Theorem 5.6.10. *Let $DC = 2\sigma(h/(1-\rho) + F^+_{rcv} + 2\lceil \frac{kF^+_{ack}}{h/(1+\rho)} \rceil F^+_{ack}) \cdot \frac{1+\rho}{1-\rho} + 2\sigma(F^+_{rcv} + kF^+_{ack})$. γ satisfies the progress property with $x_p = min(R_i, R_j) - 2\sigma(\frac{h}{1-\rho} + F^+_{rcv}) - DC$, $\alpha'_p = \frac{h}{1-\rho} + F^+_{rcv}$, and $\alpha''_p = \frac{h}{1-\rho} + F^+_{rcv}$.*

Proof. Without loss of generality, suppose two nodes p_i and p_j are within distance x_p at real time t'. $dist(p_i, p_j) \leq x_p$ at time t' guarantees that p_i and p_j remain in each other's communication radius for at least the amount of time $h/(1-\rho) + F^+_{rcv}$ in real time even though both p_i and p_j moves at a maximum speed (σ) away from each other. Applying Lemma 5.6.9, we get that p_i and p_j receives each other's hello message during $[t', t' + h/(1-\rho) + F^+_{rcv}]$ in real time. Furthermore, $dist(p_i, p_j) \leq x_p$ at time t' also guarantees that by receiving the hello message during $[t', t' + h/(1-\rho) + F^+_{rcv}]$ in real time, the call to closeEnough() returns true causing the link state to be LU if the link state was not LU and the link state to remain in LU otherwise.

Now, without loss of generality, suppose that p_i and p_j are within distance x_p for all times during $[t, t+\alpha_p]$ in real time. By the above argument, it is clear that at real time $t + h/(1-\rho) + F^+_{rcv} (= t + \alpha'_p)$ the link state of both p_i and p_j are LU. Also, from real time $t + \alpha_p$ to real time $t + \alpha_p + \alpha''_p$, each call to closeEnough() will return true at both p_i and p_j. When closeEnough() returns true at p_i (resp. p_j), $sendUntil_i[j]$ (resp. $sendUntil_j[i]$) is set to at least SU_2. Considering the worst case, the amount of local time SU_2 can be at least $SU_2/(1+\rho) = \alpha''_p$ in real time. Also, considering the worst case, closeEnough() might have returned true at time $t + \alpha_p$. Hence, until real time $t + \alpha_p + SU_2/(1+\rho) = t + \alpha_p + \alpha''_p$, $LS_i(j) = LU$ and $LS_j(i) = LU$. Therefore, we have $LS^{t_1}_i(j) = LU$ and $LS^{t_1}_j(i) = LU$ for all $t_1 \in [t + \alpha'_p, t + \alpha_p + \alpha''_p]$ in real time. □

The final theorem shows that the reliable delivery property holds.

Theorem 5.6.11. *γ satisfies the reliable delivery property.*

Proof. The AML guarantees the *no duplication* property. This property says, for a node p_k, no two receive events at p_k are caused by the same broadcast event. Since the ND/P2P layer simply forwards application messages to the AML, we can say that,

for a node p_k, no two application receives are caused by the same application send. So, for the reliable delivery property, it only remains to prove that an application message sent from p_i to p_j while p_i's link state is LU with respect to p_j is received by the application layer of p_j.

Without loss of generality, suppose node p_i sends an application message at local time t to node p_j while $LS_i^t(j) = LU$. Notice how $sendUntil_i[j]$ and $recvUntil_i[j]$ is set by lines 13 and 15. By the values of SU_1, RU_1, SU_2, and RU_2, $sendUntil_i[j]$ and $recvUntil_i[j]$ satisfies $sendUntil_i^{t'}[j] + (1+\rho)(kF_{ack}^+ + 2\lceil \frac{kF_{ack}^+}{h/(1+\rho)} \rceil F_{ack}^+ \cdot \frac{1+\rho}{1-\rho}) \leq recvUntil_i^{t'}[j]$ for all local time t'. If the last update to $sendUntil_i[j]$ before local time t was by calling updateDeadlines() at line 15, then Lemma 5.6.7 implies that, starting from local time $sendUntil_i^t[j]$, during the next $2\lceil \frac{kF_{ack}^+}{h/(1+\rho)} \rceil F_{ack}^+ \cdot \frac{1+\rho}{1-\rho} + kF_{ack}^+$ amount of time in real time, p_i and p_j will be within each other's communication radius. Even if the the most recent update to $sendUntil_i[j]$ before local time t was by calling updateDeadlines() at line 13, we still have that starting from local time $sendUntil_i^t[j]$, during the next $2\lceil \frac{kF_{ack}^+}{h/(1+\rho)} \rceil F_{ack}^+ \cdot \frac{1+\rho}{1-\rho} + kF_{ack}^+$ amount of time in real time, p_i and p_j will be within each other's communication radius. This is because formula SU_1 guarantees that $sendUntil_i^t[j]$ is set to a local time value such that the real time left for p_i to reach the local time $sendUntil_i^t[j]$ is less than or equal to the real time left for p_j to reach its local time $sendUntil_j^{t_j}[i]$ where t_j is the local time of p_j that is equal to p_i's local time t when both are transformed to real time (see the proof of Theorem 5.6.8 (Case 2)).

Now, recall that kF_{ack}^+ is the upper bound on the total real time of emptying the the k-bounded buffer if the buffer is full (when there is no control (hello) message interference) and $2\lceil \frac{kF_{ack}^+}{h/(1+\rho)} \rceil F_{ack}^+ \cdot \frac{1+\rho}{1-\rho}$ is the upper bound on the total real time of hello message transmissions interfering with k consecutive application messages in the k-bounded send buffer. Thus, a message already in the k-bounded

send buffer is guaranteed to be sent to the AML within the amount of real time $kF^+_{ack} + 2\lceil \frac{kF^+_{ack}}{h/(1+\rho)} \rceil F^+_{ack} \cdot \frac{1+\rho}{1-\rho} - F^+_{ack}$. From the previous paragraph and by Lemma 5.6.5, we therefore have that the ND/P2P layer of p_j receives the application message sent by p_i while $LS^t_i(j) = LU$.

It remains to show that the application layer of p_j receives the message by p_i. Considering p_i, if $sendUntil^t_i[j]$ was set by calling updateDeadlines() at line 15 at real time t'' (p_i's local time t in real time corresponds to t''), then by Lemma 5.6.7 and 5.6.9, p_j will receive $hello_i$ from p_i until $t'' + h/(1-\rho) + F^+_{rcv}$ in real time. Upon p_j receiving $hello_i$ from p_i at p_j's local time t'_j, the call to updateDeadline() at line 13 and the value of RU_1 ensures that $recvUntil^{t'_j}_j[i]$ in real time is greater than or equal to p_i's local time $sendUntil^t_i[j]$ in real time plus $kF^+_{ack} + 2\lceil \frac{kF^+_{ack}}{h/(1+\rho)} \rceil F^+_{ack} \cdot \frac{1+\rho}{1-\rho}$ in real time. Hence, for all $t_1 \in [t'_j, recvUntil^{t'_j}_j[i])$ in p_j's local time, $LS^{t_1}_j(i) \neq LD$ holds. Thus, by lines 37, 43, and 29, the application message sent by p_i will be delivered to p_j's application layer. Now, if $sendUntil^t_i[j]$ was set by calling updateDeadlines() at line 13 at real time t'' (p_i's local time t in real time corresponds to t''), then, by formula SU_1, $sendUntil^t_i[j]$ in real time is less than $sendUntil^{t''_j}_j[i]$ in real time where p_i's local time t and p_j's local time t''_j have the same time value when both are transformed to real time. Since $sendUntil^{t'}_j[i] + (1+\rho)(kF^+_{ack} + 2\lceil \frac{kF^+_{ack}}{h/(1+\rho)} \rceil F^+_{ack} \cdot \frac{1+\rho}{1-\rho}) \leq recvUntil^{t'}_j[i]$ holds for all local times t' and $LS^{t_2}_j(i) \neq LD$ holds for all local times $t_2 \in [sendUntil^{t'}_j[i], recvUntil^{t'}_j[i])$, lines 37, 43, and 29 guarantee that the application message sent by p_i will be delivered to p_j's application layer. □

5.7 Discussion

In this section, we apply parameter values to obtain the maximum distance between two nodes that guarantee link up with each other and compare the results of different parameter values.

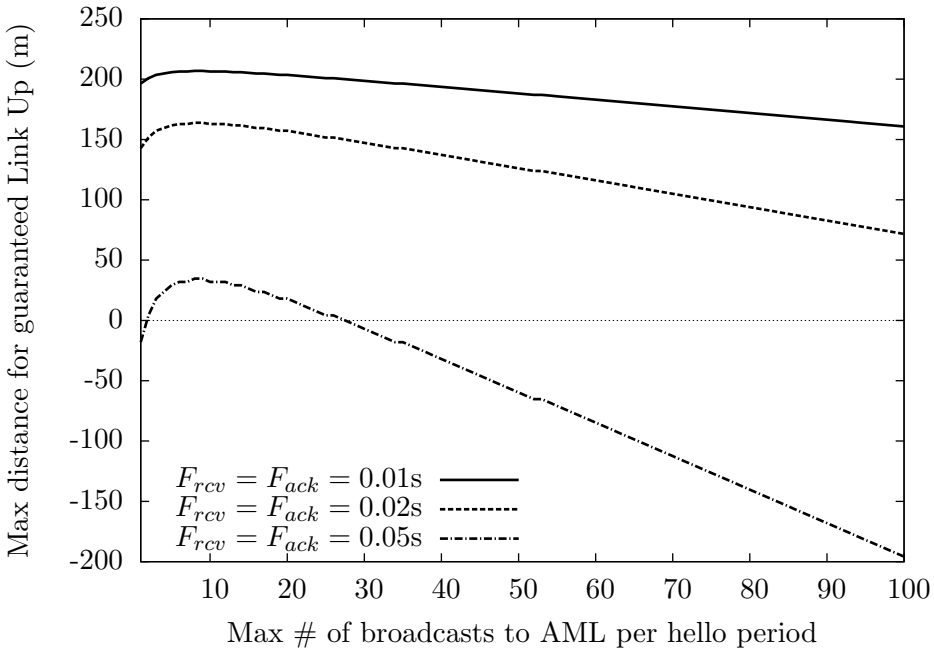


Figure 5.4: Max distance between two nodes for guaranteed link up with each other (x_p of Theorem 5.6.10) when max speed is 50km/h.

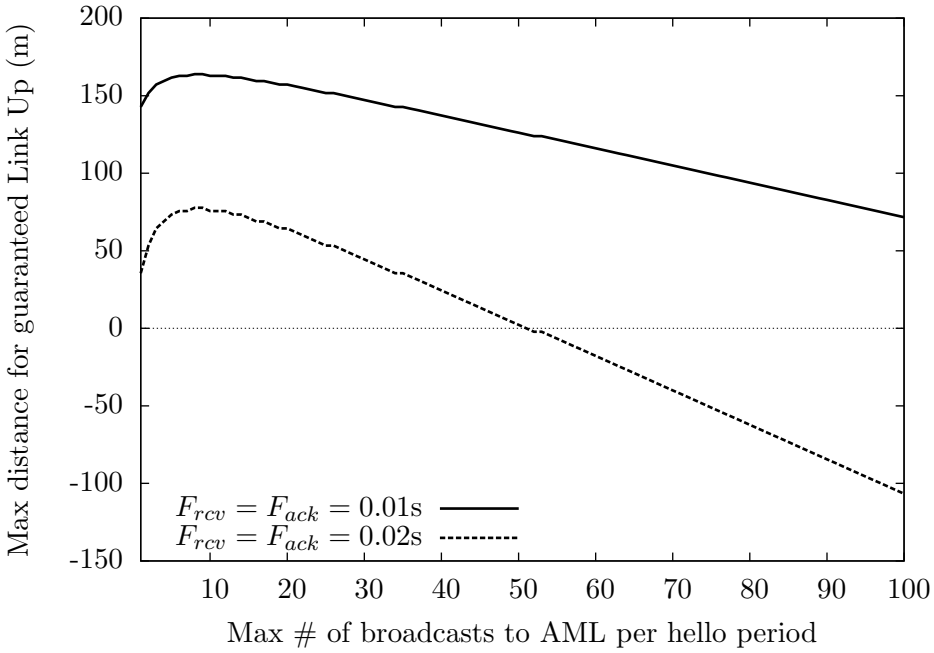


Figure 5.5: Max distance between two nodes for guaranteed link up with each other (x_p of Theorem 5.6.10) when max speed is 100km/h.

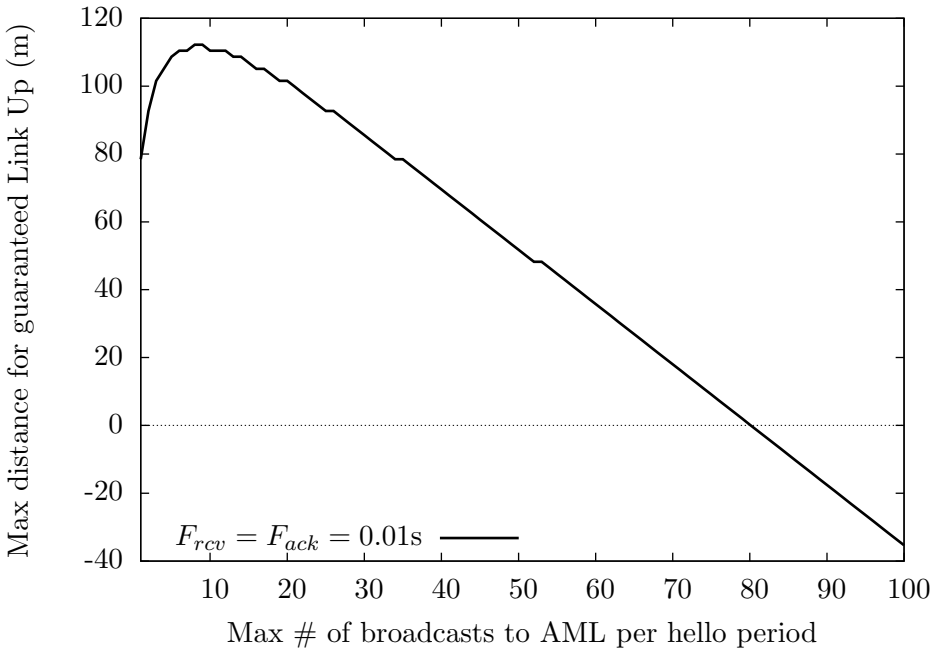


Figure 5.6: Max distance between two nodes for guaranteed link up with each other (x_p of Theorem 5.6.10) when max speed is 160km/h.

Figures 5.4, 5.5, and 5.6 show the x_p value (the maximum distance between two nodes to guarantee link up with each other) of Theorem 5.6.10 for different F^+_{rcv} and F^+_{ack} values and different σ values by increasing the maximum number of broadcasts to the AML per hello period (parameter b; see Section 5.5) given $R = 250$ meters (typical value for IEEE 802.11 [29, 56]), $k = 110$ (maximum size of the send buffer), $\rho = 10^{-6}$, and $h = (b+2)F^+_{ack} \cdot \frac{1+\rho}{1-\rho}$.

Figure 5.4 shows the x_p value by ranging b from 1 to 100 where nodes move in a maximum speed of 50km/h (an urban area considering vehicles as nodes). Expecting that F^+_{rcv} and F^+_{ack} to be tens of milliseconds (ms) and assuming that $F^+_{rcv} = F^+_{ack}$, we have considered three cases for the values of F^+_{ack}: 10ms, 20ms, and 50ms. For the cases of 10ms and 20ms, the x_p values stay above 160 meters and 71 meters, respectively, even though $b = 100$. For the case of 50ms, the x_p value hits its

maximum value 34.7203 meters when $b = 8$ and we also get that $x_p > 0$ only when b ranges from 2 to 27. So, Figure 5.4 shows that a large x_p value can be obtained even though having a large b, when F^+_{ack} ranges from 10ms to 20ms. The reason why the x_p value first increases and then gradually decreases as b grows is as follows: In our algorithm, when node p_i's link state becomes link up with respect to node p_j by calling closeEnough(), we require that the next hello message from p_j to be received by p_i. So, as the hello period h grows large (=as b grows large) p_i and p_j have to be closer to each other to guarantee link up with each other. However, as h grows large (=as b grows large), there will be fewer hello message broadcasts interfering with application message broadcasts which means that p_i and p_j can be farther away from each other but still guarantee link up with each other. The increase and decrease of the x_p value is showing the result of this trade-off.

Figure 5.5 shows the case when nodes move in a maximum speed of 100km/h (a regular highway considering vehicles as nodes). In this case, even $F^+_{rcv} = F^+_{ack} = 10$ms is still suitable in obtaining a large value of x_p for large values of b. Considering an even higher maximum speed of 160km/h (the autobahn considering vehicles as nodes), Figure 5.6 shows that even $F^+_{rcv} = F^+_{ack} = 10$ms is not suitable in obtaining a large value of x_p for large values of b. However, for a small value of b (≤ 50), $F^+_{rcv} = F^+_{ack} = 10$ms is still suitable in getting a large value of x_p.

6. REGIONAL CONSECUTIVE LEADER ELECTION IN MOBILE AD HOC NETWORKS

We consider leader election in a fixed geographic region R with bounded communication diameter which allows leader election to be performed among nodes that are relatively close to each other (leader election is performed in a *local* fashion). By bounded communication diameter we mean that if a node in R initiates the propagation of a message at some time t, then the message will be relayed through a fixed maximum number of hops D to any node that stays in the region sufficiently long after time t — this is formally defined as D-connectedness in Section 6.3.

In this section, we solve the *Regional Consecutive Leader Election* (RCLE) problem which was originally introduced in [16]. Roughly speaking, the RCLE problem requires the following:

(*Agreement*) All nodes in the region that elect a leader elect the same leader.

(*Termination*) If some live[1] node p remains in the region for a sufficiently long period of time, then p must elect a leader.

(*Validity*) If some live node p in the region elects a leader, then that leader node must have been in the region in the recent past.

(*Stability*) If some live node p in the region stops considering some other node q as the leader, then q has either crashed or left the region in the recent past.

We provide a lower bound with respect to the (*Termination*) property of the

Part of this section is reprinted from the following paper: Hyun Chul Chung, Peter Robinson, and Jennifer L. Welch, "Optimal regional consecutive leader election in mobile ad-hoc networks," In proceedings of the 7th ACM SIGACT/SIGMOBILE Workshop on Foundations of Mobile Computing (FOMC 2011), ©2011 ACM, Inc. Reprinted by permission. http://doi.acm.org/10.1145/1998476.1998485.

[1]Live nodes are those nodes that have not yet crashed or have already recovered from a crash. See Section 6.3.

RCLE problem. More specifically, we prove that any algorithm requires $\Omega(Dn)$ rounds for electing a leader, where D is the bounded communication diameter of the network and n is the total number of nodes in the system. Also, we provide a novel algorithm that solves the RCLE problem. The provided algorithm guarantees termination in $\mathcal{O}(Dn)$ rounds which shows that our algorithm is asymptotically tight with respect to time complexity. Our algorithm does not rely on the knowledge of the number of nodes in the system nor on a common start-up time.

Since the proposed algorithm solves the RCLE problem under the assumption that the region has a bounded communication diameter, the question arises how to ensure that the region has a bounded communication diameter? To answer this question, we can think of restricting the nodes to follow a certain condition on mobility. We provide a novel and intuitive condition on mobility that ensures the existence of a bounded communication diameter. The proposed mobility condition requires nodes to move in a way such that the propagated information makes a certain amount of progress towards its destination in each round while not relying on a fixed coordinate system.

6.1 Contributions

We define D-connectedness which is weaker than usual connectivity assumptions in the sense that it allows scenarios where a set of nodes within the region of interest are temporarily partitioned.

We introduce the RCLE problem by extending the leader election problem to the ever changing environment of mobile ad hoc networks.

We prove a lower bound with respect to the (Termination) property of the RCLE problem and then present and prove correct an asymptotically optimal crash fault-tolerant algorithm that solves the RCLE problem.

Finally, we provide a condition on mobility that allows us to obtain a bounded communication diameter within the region of interest.

6.2 Related Work

Geographical information has been widely used in mobile ad hoc environments (e.g., [40, 41, 39]). However, there are only a few papers that consider location information in solving leader election in MANETs. Chung et al. [16] provide an algorithm that solves the RCLE problem with a message bit complexity of $\mathcal{O}(n(\log n + \log r))$ per node per round. The algorithm of [16] runs under the assumption that nodes have access to synchronized clocks. Also, [16] provides a condition on mobility that guarantees the existence of a bounded communication diameter. However, this mobility condition becomes highly restrictive in some situations because of the fact that it relies on a fixed coordinate system. In Section 6, we significantly improve on the results of [16], by providing an algorithm that solves the RCLE problem with a message bit complexity of $\mathcal{O}(\log n + \log r)$ per node per round without requiring nodes to have access to synchronized clocks. We also provide a mobility condition that is less restrictive in the sense that it does not rely on any fixed coordinate system. In addition, we provide a lower bound on the number of rounds for a node to elect a leader.

In Kuhn et al. [45], the entire geographical space is divided into regions where nodes in a region forms a single hop network. A leader is elected for each region and these leaders form a backbone for message propagation. Our approach differs from [45], since we consider a single fixed region in which nodes exchange messages through multi-hop communication.

Hatzis et al. [33] provide an algorithm where a leader is elected by nodes encountering each other. When two or more nodes meet, they decide on which one of them

continues to participate in electing a leader. The entire space is divided into non-intersecting subspaces and nodes get to meet each other when they fall into the same subspace. Furthermore, a probabilistic analysis is given considering the movement of nodes as random walks. In Section 6.6, we provide a condition on mobility that gives a *deterministic* bound on message propagation.

Leader election algorithms presented in [9, 20, 36, 51, 52, 61] and [71] consider a mobile environment where geographical information is *not* used. In addition, they all consider networks that can have an arbitrarily large communication diameter. Our approach considers leader election in a region with a bounded communication diameter which is a better fit for situations when leader election is needed only among nearby nodes.

6.3 System Model and Problem Specification

We consider a system consisting of a set Π of nodes that move in two-dimensional Euclidean space. Each node has a unique id and communicates with other nodes via wireless broadcast.

We assume that nodes execute in synchronous rounds of communication and computation, where each round lasts Δ time, for an appropriately chosen value Δ. Such rounds can be provided by, for instance, the use of the abstract MAC layer [45]. This round abstraction allows us to focus on the algorithmic aspects; the collision-free scheduling of wireless transmissions, while nevertheless being an important problem, is assumed by the round abstraction. It is important to note that the round abstraction does *not* provide nodes with additional timing information. That is, we neither require nodes to be equipped with synchronized clocks nor assume that nodes can access the current round number.

Every node executes an instance of a distributed algorithm and is modeled as

a deterministic state machine. In more detail, each round begins with broadcasts by the nodes and continues with nodes receiving certain broadcasts. We denote the (possibly infinite) *set of messages* that can be generated by the algorithm as $\mathcal{M}$. At the end of each round, each node uses its current state and the set of messages received during the round to change its state and decide what to broadcast at the beginning of the next round. An *execution* of an algorithm is simply an infinite sequence of rounds.

Nodes can *fail* by crashing and might recover from the crash after some non-zero number of rounds; a node reinitializes its state upon recovery. We assume that processing done at the end of each round takes 0 time units and only *live* nodes—nodes that have not crashed or have already recovered—operate during a round. As we do not assume that all nodes startup at the exact same time, we simply model nodes that have not yet booted (not yet recovered) as crashed nodes.

We focus on a specific *region* R of the two-dimensional space in which the nodes are present. Every node has access to geographical information with respect to region R by using functions `EnteredSinceLastRound`() and `Location`() which can be provided by an underlying location service.

Calling function `Location`() yields the exact position in the Euclidean space of the querying node, whereas function `EnteredSinceLastRound`() evaluates to true if (a) the node has entered R since the end of the last round or (b) the node has just recovered from a crash inside R.

To specify which broadcasts are received by a node in a round, we use the notion of two nodes p and q being *connected in the round*: this means that throughout the round, the Euclidean distance between p and q is at most C, the (common) *communication radius*. If p and q are live and connected throughout a round, node p receives exactly one copy of the message that is broadcast by q at the beginning

of the round (p can also possibly receive a message from a node that is within C at some point during the round). Thus, communication is timely and reliable, with no lost, corrupted or spurious messages.

For each round r, we define the following sets of nodes, depending on their location with respect to region R:

- $\Pi_A^r = \{p_i \in \Pi \mid p_i$ is live and within R throughout round $r\}$.
- $\Pi_E^r = \{p_i \in \Pi \mid p_i$ is live and within R at the end of round $r\}$.
- $\Pi_B^r = \{p_i \in \Pi \mid p_i$ is live and within R at the beginning of round $r\}$.

For rounds r_a and r_b with $r_a \leq r_b$, we define $\Pi_A^{[r_a, r_b]}$ as the set $\{p_i \in \Pi \mid \forall r, r_a \leq r \leq r_b \colon p_i \in \Pi_A^r\}$.

Sometimes we only care about a prefix of an execution S until some round r; we denote this by $S|_r$. Executions S and S' are indistinguishable for a node p, if p receives the same messages in S and the values of `Location`() and `EnteredSinceLastRound`() are the same at p in S and S', for all rounds r. Intuitively speaking, node p observes an indistinguishable environment in both executions.

Given a particular execution, we define a *Just-In-Time (JIT) path starting at round r from node p to node q of length m* to be a sequence of nodes $p = v_0, v_1 \ldots, v_m = q$ such that, for all i, $0 \leq i < m$, $v_i \in \Pi_B^{r+i}$, v_i is live throughout $r + i$, $v_{i+1} \in \Pi_A^{r+i}$, and v_i and v_{i+1} are connected in round $r + i$.

Without any assumptions on the mobility pattern of the nodes, it is impossible to guarantee the existence of JIT paths. Thus, we are motivated to assume that every execution satisfies the following property, which was originally introduced in [16] and is also related to the "dynamic diameter" of [44].

Assumption 6.3.1 (D-Connectedness)**.** *There exists an integer D such that for every pair of nodes p and q, and every round r, the following holds: if $p \in \Pi_B^r$, p is*

live throughout r, and $q \in \Pi_A^{[r,r+D-1]}$, then there exists a JIT path starting at round r from p to q of length at most D.

Intuitively speaking, Assumption 6.3.1 guarantees the existence of a JIT path from p to q, if node q stays sufficiently long inside the region, starting at the time when p sends its message. We further assume that the value of D is known to all nodes in the system.

6.3.1 Problem Specification

We assume that every node has access to a special variable *leader* that is either set to $\perp$ or contains a node id. We denote the value of *leader* at node p_i at the end of round r as $leader_{p_i}^r$. If p_i has crashed before or in round r and has not yet recovered (or has not yet booted), we assume that $leader_{p_i}^r = \perp$.

Definition 6.3.2. *An algorithm A solves the* regional consecutive leader election (RCLE) problem *if there exist integer bounds B_T, B_V, and B_S such that the following properties are satisfied for every D-connected execution of A:*

$$\forall r \in \mathbb{N}\ \forall p_i, p_j \in \Pi_E^r\colon \Big(\Big(leader_{p_i}^r \neq \perp \neq leader_{p_j}^r \Big) \Rightarrow \Big(leader_{p_i}^r = leader_{p_j}^r \Big) \Big) \qquad (Agreement)$$

$$\forall r \in \mathbb{N}\ \forall p_j \in \Pi\colon \Big(\big(\exists p_i \in \Pi_E^r\colon leader_{p_i}^r = j \big) \Rightarrow \Big(\exists r' \in [r - B_V, r]\colon \Big(p_j \in \Pi_B^{r'} \wedge leader_{p_j}^{r'} = j \Big) \Big) \Big) \qquad (Validity)$$

$$\forall r \in \mathbb{N}\ \forall p_i \in \Pi\colon \Big(\big(\forall r_1 \in [r, r + B_T] : p_i \in \Pi_A^{r_1} \big) \Rightarrow \big(\exists r_2 \in [r, r + B_T] : leader_{p_i}^{r_2} \neq \perp \big) \Big) \qquad (Termination)$$

$$\forall r_1, r_2 \in \mathbb{N}\ \forall p_j \in \Pi\colon \Big(\Big(r_1 < r_2 \wedge p_i \in \Pi_A^{[r_1,r_2]} \wedge leader_{p_i}^{r_1} = j \wedge leader_{p_i}^{r_2} \neq j \Big) \Rightarrow (\exists r \in [r_1 - B_S, r_2] : p_j \notin \Pi_A^r) \Big) \qquad (Stability)$$

Property (*Validity*) ensures that only nodes, which have recently been inside the region (i.e. within bounded time) and have claimed to be the leader, can be elected as the leader, whereas (*Stability*) guarantees that the leader only changes when the previous leader node crashed or is no longer in the region. Note that (*Termination*) only requires a node to elect a leader *if* it remains in the region for a sufficiently long period of time.

6.4 A Lower Bound on Time Complexity

In this section, we will prove a lower bound on the time (i.e. number of rounds) that it takes until a newly incoming node has set its leader variable in the worst case.

Lemma 6.4.1. *Let A be an algorithm that solves the RCLE problem. Then there is an execution such that the following hold:*

(a) For all $r \in [1, D)$ and for all $p_i \in \Pi_A^{[1,D)}$ we have that $leader_{p_i}^r = \perp$;

(b) $|\Pi_A^{[1,\infty)}| = \Theta(n)$.

Proof. Assume in contradiction that no such execution exists. First, consider the execution S where all nodes in Π are in $\Pi_A^{[1,\infty)}$ and are all pairwise disconnected during $[1, D)$. Moreover, in round D, the connectivity graph of nodes in the region is fully connected, and from round $D+1$ on, the connectivity graph is arbitrary, but with the restriction that D-connectedness holds. By assumption, the set of nodes $P \subseteq \Pi_A^{[1,D]}$, such that every $p_i \in P$ claims to be the leader in some round during $[1, D)$ in S, is nonempty. Due to (*Stability*), every node $p_i \in P$ still has $leader_{p_i}^D = i$, which, by (*Agreement*), implies that $|P| = 1$, i.e., $P = \{p_j\}$, for some node p_j. Now consider the execution S' that is identical to S for all nodes in $\Pi \setminus \{p_j\}$ and where p_j is not inside the region during $[1, D]$. Clearly S' is indistinguishable from S for all nodes in $\Pi \setminus \{p_j\}$ until round D, and thus no node inside the region terminates before round D. Note that execution S' satisfies (a) and (b), therefore providing a

contradiction. □

Theorem 6.4.2. *Let A be an algorithm that solves the RCLE problem and suppose that A satisfies termination with some bound B_T. Then it holds that $B_T = \Omega(Dn)$.*

Proof. We will show by induction that there exists a permutation π, a sequence of executions $(S_k)_{1 \leq k \leq n}$, and a sequence of rounds $(r_k)_{1 \leq k \leq n}$, such that node $p_{\pi(i)}$ takes $r_i \geq iD$ rounds before setting its leader variable in execution S_i, for $1 \leq i \leq n$.

Lemma 6.4.1 shows that there is some execution S_1 where no node terminates before round D. Since algorithm A is correct, some node $p_{\pi(1)}$ is the first one to claim to be the leader in some round $r_1 \geq D$ in S_1; this provides us with the induction base. Considering property (*Validity*) we have that

$$\forall r \in [1, r_1)\ \forall p_j \in \Pi_A^r \colon leader_{p_j}^r = \perp\ .$$

Moreover, no other node apart from $p_{\pi(1)}$ can set its leader variable in round r_1. To see why this is the case, assume that $p_{\pi(1)}$ crashed at the beginning of round r_1 and some node p' has $leader_{p'}^{r_1} = \pi(1)$. Clearly, since $p_{\pi(1)}$ crashed and thus never claimed to be the leader, we have a violation to (*Validity*).

Note that we can assume that $p_{\pi(1)}$ is connected to every other node in round r_1. We will make use of this assumption in the induction step below.

For the induction step, suppose that, for some $i \in [1, n)$, node $p_{\pi(i)}$ has claimed to be the leader in round r_i, i.e., $leader_{p_{\pi(i)}}^{r_i} = i$, and assume that nodes in

$$P = \Pi \setminus \{p_{\pi(1)}, \ldots, p_{\pi(i)}\}$$

have not yet set their leader variable by round r_i in execution S_i. By using the same argument as for $p_{\pi(1)}$ in the induction base, we can assume that only $p_{\pi(i)}$ sets its

leader variable in round r_i. More specifically, the induction hypothesis tells us that

$$\forall r \in [1, r_i] \supseteq [1, iD] \colon leader^r_{p_j} = \perp \ ,$$

for all nodes $p_j \in P$, and that $p_{\pi(i)}$ is connected to every node in $\Pi^{r_i}_A$ during round r_i. Now consider the following execution S_{i+1}: During rounds $[1, r_i)$, execution S_{i+1} is equivalent to the prefix $S_i|_{[1,r_i)}$, but we assume that node $p_{\pi(i)}$ exits the region at the end of round r_i in S_{i+1}, just before electing itself as the leader. Furthermore, if $i > 1$, node $p_{\pi(i-1)}$ reenters the region at the end of r_i. The nodes in P remain in the region forever in execution S_{i+1}. Let $\pi(i+1)$ be such that $p_{\pi(i+1)}$ is the first node that claims to be the leader in S_{i+1} in some round $r_{i+1} > r_i$.

We will now show that $r_{i+1} - r_i \geq D$. Suppose that this is false and consider execution S'_{i+1} that is identical until round $r_i + D$ to S_{i+1}, for all nodes except for $p_{\pi(i)}$. The only difference is that $p_{\pi(i)}$ does not leave the region, but instead is disconnected from all other nodes during rounds $(r_i, r_i + D)$. Note that this does not violate D-connectedness, since by assumption $p_{\pi(i)}$ is connected to every node in $\Pi^{r_i}_A$ in r_i and we can assume that execution S'_{i+1} is chosen in a way such that this is also true in $r_i + D$. Considering that S'_{i+1} is indistinguishable for p_{i+1} from execution S_{i+1} up to round $r_i + D - 1 \geq r_{i+1}$, node p_{i+1} must claim to be the leader in round r_{i+1}, and since (*Stability*) holds, p_i also still claims to be the leader in r_{i+1}. As this would be a contradiction to (*Agreement*), it follows that $r_{i+1} - r_i \geq D$.

By the induction hypothesis and the fact that A satisfies (*Validity*), we have that

$$\forall r \in [1, r_{i+1}] \supseteq [1, (i+1)D] \colon leader^r_{p_j} = \perp \ ,$$

for any node $p_j \in P \setminus \{p_{\pi(i+1)}\}$ in execution S_{i+1}.

This proves that some node $p_{\pi(n)}$ does not terminate before round nD, as required.

□

6.5 An Optimal RCLE Algorithm

The pseudocode of the optimal RCLE algorithm is given in Figures 6.1 and 6.2. We will first describe the building blocks of the algorithm and then take a closer look at the election process. We denote the value of variable *var* at node p_i at the end of round r as $var^r_{p_i}$.

6.5.1 Variables and Timers

Since nodes do not have access to clocks, each node employs a local counter *enteredSince* for measuring the number of rounds that have passed since the node entered the region. On the other hand, local variable $attemptNum$ represents each node's view of the number of times that nodes have attempted to elect a leader.

The algorithm makes heavy use of *timers*, which are local counter variables with a special interface that can easily be implemented in any round-based system. By executing `startTimer`$(T, offset)$ in round r, a node p_i can set its timer T to expire in $offset$ rounds. That is, the predicate expired(T) will be true in round $r+offset$ at p_i and any following round until some other timer has been started.[2] Also, from the call of `startTimer`$(T, offset)$ on until round $r+offset-1$, the predicate `running`(T) will hold at p_i. Moreover, the call of `startTimer`$(T, offset)$ will stop any other timer T' at p_i from running (and expiring).

6.5.2 Message Fields

A node p_i communicates with other nodes by broadcasting messages consisting of 5 fields. If the *oid* field (see below) of the message contains the id of node p_g, we say

[2]Recall that p_i does not know the actual round number, as we do not assume access to clocks.

```
   ⟨ Variables and Initialization ⟩
 1: leader, candidate ←⊥; in Π ∪ {⊥}
 2: enteredSince, longest ← −1; in ℤ
 3: attemptNum ← 0; in ℕ
 4: nullMsg ← ⟨I, 0, 0, 3D, i⟩; in M; constant
 5: toSend, prevToSend ← nullMsg; in M
 6: Timer waitT, electT, leaderT;
    ---------------------------------------------------------------
 7: if EnteredSinceLastRound() then
 8:    stop all timers;
 9:    initialize variables (as above);
10:    startTimer(waitT, 5D);

11: enteredSince ← enteredSince + 1;
12: incTimeStamps(recBuf ∪ {toSend, prevToSend});
13: longest ← max(longest+1, {m.entSince | m ∈ recBuf});
14: msg ← max{m ∈ (recBuf ∪ nullMsg)};

15: if msg > toSend then
16:    toSend ← msg;
17:    attemptNum ← msg.attNum;
18:    if ¬running(waitT) then
19:       if msg.type = L then
20:          leader ← msg.oid;
21:          startTimer(leaderT, 2D − msg.age);
22:       else
23:          leader ←⊥;
24:          if msg.oid = i then
25:             candidate ←⊥;
26:          else
27:             candidate ← msg.oid;
28:          startTimer(electT, 2D − msg.age);
```

Figure 6.1: Optimal RCLE algorithm; code for node p_i (part 1 of 2).

```
29: if expired(waitT) then
30:   if (longest > enteredSince) ∧ differ(prevToSend, toSend) then
31:     longest ← enteredSince;
32:     prevToSend ← toSend;
33:     startTimer(waitT, 5D);
34:   else
35:     if toSend.type = L then
36:       if toSend.oid ≠ i then
37:         leader ← toSend.oid;
38:       startTimer(leaderT, 2D − toSend.age);
39:     else
40:       startTimer(electT, 2D − toSend.age);

41: if expired(electT) then
42:   if candidate = i then
43:     leader ← i;
44:     attemptNum ← attemptNum + 1;
45:     toSend ← ⟨L, attemptNum, enteredSince, 0, i⟩;
46:     startTimer(leaderT, D);
47:   else
48:     startTimer(leaderT, 3D − toSend.age);

49: if expired(leaderT) then
50:   if leader = i then
51:     toSend ← ⟨L, attemptNum, enteredSince, 0, i⟩;
52:     startTimer(leaderT, D);
53:   else
54:     startInstance();

55: if toSend ≠ nullMsg then
56:   broadcast toSend at the start of next round;

57: procedure startInstance()
58:   leader ←⊥;
59:   candidate ← i;
60:   attemptNum ← attemptNum + 1;
61:   toSend ← ⟨I, attemptNum, enteredSince, 0, i⟩;
62:   startTimer(electT, 2D);
```

Figure 6.2: Optimal RCLE algorithm; code for node p_i (part 2 of 2).

that p_g has *generated* the message. Considering some round r_s, we will now describe the fields of a message m that was generated (i.e. initially broadcast) by node p_g in round r_g and which is broadcast in round $r_s > r_g$ by some node p_s:

1. *type*: We distinguish between so called *leader messages* ($m.type = \text{L}$), which are generated only by nodes that claim to be the leader, and *instance messages* ($m.type = \text{I}$).
2. $attNum$: This field contains the value of $attemptNum$ at generator p_g in round r_g.
3. $entSince$: The time stamp $entSince$ corresponds to the number of rounds that have passed since p_g has entered the region, assuming it has not yet exited. That is, $m.entSince = enteredSince_{p_g}^{r_s}$.
4. *age*: The value of $m.age$ is the number of rounds since the message has been generated, i.e., $m.age = r_s - r_g$.
5. *oid*: This field simply contains the id of the generator, i.e., $m.oid = g$.

Note that the values of fields $m.entSince$ and $m.age$ need to be adapted in every round, since p_g continuously increases its variable $enteredSince$ as long as it is in the region. This is ensured by calling the function `incTimeStamps`(M) in Line 12, which takes a set M of messages and increases the value of $m.entSince$ and $m.age$ by 1, for every $m \in M$. For example, if the message $\langle I, a, e, 0, i\rangle$ is generated in round r by node p_i, its fields will have the values $\langle I, a, e + k, k, i\rangle$ by round $r + k$. When we consider a message m with the field values of some specific round r, we denote this as m^r. We will omit the superscript r, when it is clear from the context.

6.5.3 Priority and Similarity of Messages

As we aim for low message complexity, it is important to ensure that relevant information is still propagated within the region. To this end, we introduce a rela-

tion on messages, which is first used by the algorithm to select the most "important" received message msg in Line 14 and then in Line 15, in order to check if the information contained in msg is relevant for the node.

We say that *message* $m_1 = \langle t_1, a_1, e_1, r_1, i\rangle$ *has priority over message* $m_2 = \langle t_2, a_2, e_2, r_2, j\rangle$, denoted by $m_1 > m_2$, *iff* one of the following conditions hold:

1. $(t_1 = \mathrm{L}) \wedge (t_2 = \mathrm{L}) \wedge ((a_1 > a_2) \vee ((a_1 = a_2) \wedge (r_1 < r_2)) \vee ((a_1 = a_2) \wedge (r_1 = r_2) \wedge (i > j)))$.
2. $(t_1 = \mathrm{I}) \wedge (t_2 = \mathrm{L}) \wedge (a_1 > a_2)$.
3. $(t_1 = \mathrm{L}) \wedge (t_2 = \mathrm{I}) \wedge (a_1 \geq a_2)$.
4. $(t_1 = \mathrm{I}) \wedge (t_2 = \mathrm{I}) \wedge ((a_1 > a_2) \vee ((a_1 = a_2) \wedge (e_1 > e_2)) \vee ((a_1 = a_2) \wedge (e_1 = e_2) \wedge (r_1 > r_2)) \vee ((a_1 = a_2) \wedge (e_1 = e_2) \wedge (r_1 = r_2) \wedge (i > j)))$.

For two messages of different message types, the message that has a higher value in its $attNum$ field has higher priority than the other; if even the values in their $attNum$ fields are the same, then the leader message takes priority over the instance message. For two leader messages, the message that has a higher value in its $attNum$ field has higher priority than the other; if the values in their $attNum$ fields are the same, then the one that was generated *later* has priority over the other; if the two messages were even generated at the same round, then the one with the higher *oid* takes priority over the other. Finally, for two instance messages, they simply follow lexicographical ordering.

Furthermore, messages m_1 and m_2 are called *similar*, if they are either identical or both leader messages that were generated by the same node, having the same value in the field $attNum$. Predicate `differ`(m_1, m_2) (see Line 30) holds if m_1 and

m_2 are not similar, formally,

$$\texttt{differ}(m_1, m_2) \equiv (m_1 \neq m_2) \wedge \neg (m_1 = \langle L, a, \_, \_, j\rangle \wedge m_2 = \langle L, a, \_, \_, j\rangle)\,.$$

We will make use of this notion of similarity for determining when to exit the waiting phase (see Section 6.5.4.1).

6.5.4 Description of the Election Process

Depending on which timer is running (*waitT*, *electT*, or *leaderT*) and the state of variable *leader*, we distinguish nodes to be in one of 3 phases: waiting phase, election phase, or leader phase.

6.5.4.1 Waiting Phase

When a node p_i enters the region, it resets its state to the initial configuration (Line 7 et seq.). It also sets its timer *waitT* to expire in $5D$ rounds (Line 10) and thus starts *waiting* by entering the so called initial waiting phase. Moreover, p_i marks the round when it has entered by setting its variable *enteredSince* to 0, which will be incremented until p_i leaves the region. In every computing step, p_i chooses the greatest received message (Line 14) with respect to the priority relation and updates its variables *toSend* and *attemptNumber* accordingly. Node p_i also stores the maximum value of the *entSince* fields of the received messages in its variable *longest*, which corresponds to the current value of variable *enteredSince* of the node that has been in the region longest. While p_i is in the waiting phase, it restricts itself on forwarding the message stored in *toSend* and performs no computation otherwise. When the timer *waitT* expires for the first time, p_i checks if itself has entered the region earliest and if the messages stored in variables *prevToSend* and *toSend* are not similar. If some node has entered before p_i did, p_i will restart *waitT* and starts

another waiting phase of length $5D$ rounds. This process will repeat itself until either (a) all previously entered nodes have left the region, or (b) p_i has received similar messages in two consecutive waiting phases. Note that the latter will happen when some (previously entered) node claims to be the leader.

6.5.4.2 Election Phase

When p_i does not reset its timer $waitT$, it sets its timer $leaderT$ or $electT$ instead, the expiration of which depends on the message stored in p_i's *toSend* variable; we call nodes where $\neg$`running`($waitT$) is true, *"non-waiting"* nodes. If p_i, while running either timer $leaderT$ or $electT$, receives an instance message m_1 that has higher priority than the stored message in its variable *toSend*, then p_i stops the currently running timer and starts timer $electT$ with the expiration round being set to $2D - m_1.age$ rounds after the current round. This allows nodes to synchronize their $electT$ timers based on the highest priority message in R. In the round when $leaderT$ expires, p_i calls `startInstance`(), assuming that p_i has not received a leader message from some other node so far. In `startInstance`(), node p_i sets its $electT$ timer, generates an instance message m_i and listens for (higher priority) messages. In the round when $electT$ expires, p_i checks if variable *toSend* still contains message m_i. If so, node p_i concludes that there are no other nodes in the region that have priority and elects itself as the leader. Otherwise, p_i waits for a leader message from the node whose id is stored in variable *toSend* (by starting timer $leaderT$) and updates its leader variable upon reception of such a message.

6.5.4.3 Leader Phase

If p_i itself is the leader, it periodically generates leader messages by expiring and resetting the timer $leaderT$ every D rounds, until it leaves the region (or crashes). If, on the other hand, p_i assumes that some node p_j is the leader, it simply waits for

leader messages from p_j by periodically setting $leaderT$. Should p_j leave the region, p_i will call `startInstance`() (Line 54) upon expiring timer $leaderT$, and return to the election phase.

6.5.5 Proof Outline

In this section, we give a brief intuition why our algorithm is correct, i.e., satisfies Definition 6.3.2.

When a node p receives a leader message m and therefore updates its leader variable, it sets timer $leaderT$ to expire exactly $2D$ rounds after m was generated. If p receives no further leader message, its timer $leaderT$ will expire and it will reset its leader variable; thus (validity) holds (see Lemma 6.5.7).

For (agreement) (see Lemma 6.5.8), we first observe that the message priority relation ensures correct propagation of the highest priority message (Lemma 6.5.4), which ensures that no two nodes will contain distinct messages in their *toSend* variable after the election phase. Moreover, if some node claims to be the leader, no higher priority instance messages have been created in the near past (Lemma 6.5.5) or will be created while the leader does not change (Corollary 6.5.6).

Corollary 6.5.6 is also instrumental for showing (stability) (see Lemma 6.5.11), since it is used in the proof of the crucial Lemma 6.5.10, which states that the leader node does not change unless it crashes or leaves the region.

The proof of the (termination) (see Lemma 6.5.22) property consists of two parts: First, we show that if some node p remains in region R for a sufficiently long amount of time (in the worst case $15D + 10(n-1)D$ rounds where n is the total number of nodes), then p must exit its waiting phase, i.e., $\neg$`running`($waitT$) holds at p. The basic idea behind the proof of this part is that, after every sequence of $10D$ rounds, if node p decides to remain in the waiting phase, then, during those 10D rounds, there

must be a non-waiting node q that crashed or left the region. This idea is enforced by Line 30, since `differ`$(prevToSend, toSend)$ becomes true when p decides to stay in the waiting period; if, on the other hand, q elected itself as the leader and stayed alive in R long enough, then `differ`$(prevToSend, toSend)$ will become false, thus causing p to stop waiting. Even if q reenters region R after leaving R, q will not directly cause p to remain in its waiting phase since the condition $longest > enteredSince$ in Line 30 will evaluate to false due to $enteredSince_q < enteredSince_p$; note that in this case p was already inside R when q reenters.

For the rest of the termination proof, we show that if node p remains in region R sufficiently long enough after exiting its waiting phase (in the worst case $7D + 4(n-1)D$ rounds), then p elects a leader by setting its leader variable. The worst case scenario would be that all "better" candidate leaders leave region R or crash sparsely (i.e. one-by-one) without ever electing themselves as the actual leader and, finally, p electing itself as the leader. The basic idea behind the proof of this part is that the nodes that enter R after p exits its waiting phase will keep waiting as long as p is live and does not elect a leader (which is the case by assumption). This is true because p, along with the other non-waiting nodes in R, will generate new instance messages at least every $4D$ rounds. Our D-connectedness assumption ensures that nodes in their waiting phase receive new instance messages at least every $5D$ rounds so when Line 30 is executed, it will evaluate to true, causing these nodes to keep waiting.

6.5.6 Proof of Correctness

The following lemma follows directly by inspecting the code.

Lemma 6.5.1. *Suppose that* $p_i \in \Pi_A^{[r,r+k]}$, *for some* $k \geq 0$. *Then, for all* $r' \in [r, r+k]$,

(a) $toSend_{p_i}^{r'}.attNum = attemptNum_{p_i}^{r'}$.

(b) $attemptNum_{p_i}^{r'}$ *is non-decreasing.*

Lemma 6.5.2. *Suppose that* $p_i \in \Pi_A^{[r,r+k]}$*, for some* $k \geq 0$*. Then, for all* $r' \in [r+1, r+k]$ *where* `incTimeStamps`$(toSend_{p_i}^{r'-1}) \neq toSend_{p_i}^{r'}$*, we have that* $toSend_{p_i}^{r'} > toSend_{p_i}^{r'-1}$ *and* $toSend_{p_i}^{r'}.attNum \geq toSend_{p_i}^{r'-1}.attNum$*.*

Proof. Let $m_1 = toSend_{p_i}^{r'-1}$ and $m_2 = toSend_{p_i}^{r'}$. Now, suppose in contradiction, for some $r' \in [r+1, r+k]$ where

$$\texttt{incTimeStamps}(m_1) \neq m_2, \tag{6.1}$$

it holds that

$$(m_2 \leq m_1) \vee (m_2.attNum < m_1.attNum).$$

By (6.1), message m_2 must be assigned to $toSend_{p_i}^{r'}$ by either line 16, line 45, line 51, or line 61. We consider four cases:

(*Case 1*) m_2 was assigned to $toSend_{p_i}^{r'}$ by line 16: In order for m_2 to be assigned by line 16, the if-condition in line 15 must evaluate to true which directly yields a contradiction.

(*Case 2*) m_2 was assigned to $toSend_{p_i}^{r'}$ by line 45: Notice that $attemptNum$ is incremented just before executing line 45. Hence, Lemma 6.5.1.(a) tells us that $m_1.attNum < m_2.attNum$ which yields $m_1 < m_2$ by the definition of the message priority relation, a contradition.

(*Case 3*) m_2 was assigned to $toSend_{p_i}^{r'}$ by line 51: If $m_1.type = I$, then Lemma 6.5.1 implies that $m_1.attNum \leq m_2.attNum$. Hence, the definition of the message priority relation gives us $m_1 < m_2$ since $m_2.type = L$, a contradiction. If $m_1.type = L$, then, due to line 12, m_1 becomes at least 1 rounds old (i.e., $m_1.age \geq 1$). Also, by Lemma 6.5.1, we know that $m_1.attNum \leq m_2.attNum$. Hence, by the definition

of the message priority relation, we always get $m_2 > m_1$ since $m_1.age < m_2.age$ at round r' ($m_1.age \geq 1$ and $m_2.age = 0$ at round r'), a contradiction.
(*Case 4*) m_2 was assigned to $toSend_{p_i}^{r'}$ by line 61: The proof is similar to (*Case 2*). □

Lemma 6.5.3. *The following statements are true if* $leader_{p_i}^r = p_i$*:*

(a) Node p_i *did not pass the if-condition in Line 15 during rounds* $(r - 2D, r]$*.*

(b) $p_i \in \Pi_E^{r-5D}$ *and* $p_i \in \Pi_A^{(r-5D,r]}$*.*

Proof. (a). Assume in contradiction that p_i received a message m that enabled it to pass the if-condition in Line 15 in some round $r' \in (r - 2D, r]$. We divide into two cases:
(*Case 1*) `running`($waitT$) holds at p_i at round r': There must exist a round during $(r', r]$ where ¬`running`($waitT$) holds at p_i since $leader_{p_i}^r \neq \perp$. Let $r'' \in (r', r]$ be the earliest round when ¬`running`($waitT$) holds at p_i. Note that $r - r'' < 2D$. Considering that, after the generation of an instance message by p_i, it requires exactly $2D$ rounds to execute Line 43, $leader_{p_i}$ must be set to p_i using Line 20 during $[r'', r]$. This implies that p_i received a leader message m_l that was generated by itself. If m_l was generated at $r_l \in [r'', r]$, then Line 15 will prevent Line 20 from being executed since $toSend_{p_i}^{r_l}$ already contains m_l and, in any subsequent rounds after r_l, $toSend_{p_i}$ contains m_l or some message that has higher priority than m_l (by Lemma 6.5.2), a contradiction. If m_l was not generated during $[r'', r]$, then it must have been generated before or at round $r'' - 5D$ because of the fact that every node stays in their waiting phase for at least $5D$ before ¬`running`($waitT$) holds and during the waiting phase no messages are generated. This shows that $m_l.age > 2D$ at round r''. Then, at round r'', the if-condition at Line 35 evaluates to true, however, $leader_{p_i}$ will not be set by Line 37 (it will remain as $leader_{p_i} = \perp$) since $m_l.oid = p_i$, a

contradiction.

(*Case 2*) $\neg$`running`($waitT$) holds at p_i at round r': If $m.type = L$ and $m.oid = p_i$, then, similar to (*Case 1*), it must be that m was generated before the most recent round when p_i entered region R (m was generated at least $5D$ rounds before round r). Since $\neg$`running`($waitT$) holds at p_i at round r', there exists a round r_e ($\leq r'$) where $\neg$`running`($waitT$) holds at p_i at round r_e and `running`($waitT$) holds at p_i at round $r_e - 1$. Note that $m.age \leq r_e - 5D$ at round r_e. Thus, similar to (*Case 1*), at round r_e, the if-condition at Line 35 evaluates to true, however, $leader_{p_i}$ will not be set by Line 37 (it will remain as $leader_{p_i} = \perp$) since $m.oid = p_i$. This implies that $toSend_{p_i} \geq m$ by the end of round r_e. Hence, by Lemma 6.5.2 and the fact that $r' \geq r_e$, p_i cannot evaluate the if-condition in Line 15 to true, a contradiction.

If $m.type = I$ or ($m.type = L$ and $m.oid \neq p_i$), there must exist a round $r'' \in [r', r]$ where either Line 20 or Line 43 is executed. The rest of the proof is similar to (*Case* 1).

(b). The proof is immediate by inspecting the code; after node p_i enters region R, it runs its $waitT$ timer at least $5D$ rounds and while timer $waitT$ is running, p_i does not update its *leader* variable (Lines 19 to 28 are not executed). □

6.5.6.1 Agreement and Validity

Lemma 6.5.4 (Priority Propagation)**.** *Suppose that p_1 broadcasts a message m in round r. Then, for every $p_j \in \Pi_A^{[r,r+D]}$, it holds that $toSend_{p_j}^{r+D} \geq m^{r+D}$.*

Proof. For the sake of a contradiction, assume that there exists some node $p_j \in \Pi_A^{[r,r+D]}$ such that p_j has $toSend_{p_j}^{r+D} < m^{r+D}$. We know that there exists a JIT path from p_1 to p_j starting in round r; w.l.o.g., assume that this JIT path is $p_1 \to p_2 \to \cdots \to p_k \to \cdots \to p_j$, where p_k is the first node that has $toSend_{p_k}^{r+k-1} < m^{r+k-1}$. We will now show by induction that for $i \in [1, k]$, every node p_i, on the JIT path

has $toSend_{p_i}^{r+i-1} \geq m^{r+i-1}$, which, together with Lemma 6.5.2, yields the required contradiction. For the induction base, we have by assumption that $toSend_{p_1}^{r} = m^r$. For $i > 1$, the induction hypothesis tells us that $toSend_{p_{i-1}}^{r+i-2} \geq m^{r+i-2}$. Moreover, by assumption p_i has an incoming link from p_{i-1} during round $r+i-1$, thus clearly $toSend_{p_i}^{r+i-1} \geq m^{r+i-1}$. What remains to show is that for all $r' \in (r+i-1, r+D]$, we have that $toSend_{p_i}^{r'} \geq m^{r'}$. But this follows directly from the message priority relation and Lemma 6.5.2. □

Lemma 6.5.5 (Dominating Leader)**.** *Let p_ℓ be a node in $\Pi_A^{[r_\ell, r_\ell+D]}$ that generates $m_\ell = \langle L, a_\ell, \_, 0, \ell \rangle$ in round r_ℓ and suppose that $toSend_{p_\ell}^{r_\ell-1}.type \neq L$; that is, p_ℓ claims to be the new leader in round r_ℓ. Then, no instance message m_j with $a_j > a_\ell$ is generated during $[r_\ell - 3D, r_\ell + D)$.*

Proof. Assume in contradiction that some node p_j generates such a message m_j in round r_j. We divide into two cases:
(*Case 1*) $r_j \in [r_\ell - 3D, r_\ell - D]$: By Lemma 6.5.4, p_ℓ must receive a message m with

$$m.attNum \geq m_j.attNum > attemptNum_{p_\ell}$$

by round r_ℓ, contradicting Lemma 6.5.1.
(*Case 2*) $r_j \in (r_\ell - D, r_\ell + D]$: We know that $expired(leaderT)$ holds for node p_j in round r_j; let m'_j be the message that was in variable $toSend_{p_j}$ when p_j passed Line 49. Note that

$$m'_j.attNum = m_j.attNum - 1 \geq m_\ell.attNum \tag{6.2}$$

by assumption. Since p_j called `startInstance`() in Line 54, we know that $leader_{p_j} \neq j$. Thus, p_j did not set $leaderT$ using Line 52 and we can observe that timer $leaderT$

is always set such that its expiration date is at least $2D$ later than the round when m'_j was generated. It follows that m'_j was broadcast (not necessarily by p_j) during rounds $[r_j - 2D, r_j)$. By Lemmas 6.5.4, it follows that p_ℓ has set its *toSend* variable to some message $m'' \geq m'_j$ by round $r_\ell - 1$.

First consider the case where $m''.type = L$: The priority relation together with (6.2) imply a contradiction to $toSend_{p_\ell}^{r_\ell - 1} \neq L$. If $m''.type = I$, then the execution of Line 44 by p_ℓ in r_ℓ provides a contradiction to (6.2). □

Corollary 6.5.6 (Continuously Dominating Leader)**.** *Suppose that there is an integer $\lambda > 0$, such that node p_ℓ generates leader messages in every round in $\{r_\ell + kD \mid k \in [0, \lambda]\}$, which all have the fixed value a_ℓ in the field $attNum$, and suppose that $toSend_{p_\ell}^{r_\ell - 1}.type \neq L$. Then, no instance message m_j with $a_j > a_\ell$ is generated during the interval of rounds $[r_\ell - 3D, r_\ell + \lambda D)$.*

Proof. Assume in contradiction that some node generates a message m_j with $a_j > a_\ell$ in round $r \in [r_\ell - 3D, r_\ell + (\lambda - 1)D]$. The case $r \in [r_\ell - 3D, r_\ell + D]$ follows immediately by Lemma 6.5.5.

Next, we will show that the result also holds for $r \in [r_\ell + D, r_\ell + (\lambda - 1)D]$. By Lemma 6.5.4, it follows that p_ℓ will receive a message $m' \geq m_j$ by round $r + D$, causing it to pass Line 15. But since, p_i still generates a leader message m' in some round in $r_\ell + \lambda D$ with $m'.attNum = a_\ell$, this is a contradiction.

Finally, we need to consider the case where $r \in (r_\ell + (\lambda - 1)D, r_\ell + \lambda D]$, for $\lambda > 1$: Note that node p_j must have called `startInstance`() in round r after its *leaderT* timer expired. Let m' be the message that caused p_j to previously set *leaderT*, i.e., $toSend_{p_j}^{r-1} = m'$. Note that m' (or some greater message) is also received by p_ℓ. Moreover, since node p_j must have previously been in the waiting phase, we know that $p_j \in \Pi_A^{(r-5D, r]}$, i.e., p_j has received some leader message generated by

p_ℓ. Therefore, either p_ℓ passes the if-condition in Line 15 upon receiving m', and thus does not generate a leader message in round $r_\ell + \lambda D$, or p_j does not pass the if-condition in Line 15 upon receiving m', and thus does not set its timer $leaderT$ to expire according to $m'.age$. Both cases yield a contradiction. □

Lemma 6.5.7 (Validity)**.** *The algorithm in Figures 6.1 and 6.2 satisfies property (validity) with bound $B_V = 2D - 1$.*

Proof. Suppose that some node p_i has $leader^r_{p_i} = j$. This implies that p_i is no longer in the waiting phase, and thus $\neg\texttt{running}(waitT)$ holds at p_i in round r. If $i = j$, then (*Validity*) trivially holds; therefore assume that $i \neq j$. By the code, if p_i sets $leader_{p_i} \leftarrow j$ then p_i has received a leader message m generated by node p_j in round r_j, i.e., $toSend^{r_i}_{p_i} = m$, for some round r_i. Note that p_i sets its timer $leaderT$ to expire in round $r_j + 2D$. W.l.o.g., we can assume that r_i is the round where p_i updates its *leader* variable the last time before r. Again, by the code, it follows that if p_i received any message $m' > m$ during $(r_i, r]$, it would update its *leader* variable, which is a contradiction; thus we have that $toSend^r_{p_i} = m^r$. If $m^r.age \geq 2D$, then expired($leaderT$) would have been true in some round in $[r_i, r]$, and hence $leader^r_{p_i} \neq j$ by Line 58, again yielding a contradiction. On the other hand, if $m^r.age < 2D$, it follows that $r_j \geq r - 2D + 1$, which means that p_j was inside the region in round $r - 2D + 1$ or some later round, i.e., $B_V = 2D - 1$, as required. □

Lemma 6.5.8 (Agreement)**.** *The algorithm in Figures 6.1 and 6.2 guarantees property (Agreement).*

Proof. For the sake of a contradiction, assume that there is a D-connected execution where property (*Agreement*) is violated the first time in some round r_2, and let p_1

and p_2 be nodes such that $leader_{p_1}^{r_2} = v_1 \neq v_2 = leader_{p_2}^{r_2}$. Without loss of generality, let r_1 be the round when p_1 has updated $leader_{p_1}$ the last time before r_2 and assume that $r_1 \leq r_2$. Depending on how p_1 and p_2 updated their variable *leader* (Lines 20 or 43; the case of using Line 37 will be analogous to the case of using Line 20), we distinguish several cases:

(*Case 1*) Nodes p_1 and p_2 both used Line 43 (i.e., p_1 and p_2 both claim to be the leader): This implies that p_1 and p_2 both passed the check expired($electT$) in Line 41. By Lemma 6.5.3.(b), we know that $p_1 \in \Pi_A^{(r_1-2D,r_2]}$ and $p_2 \in \Pi_A^{(r_2-2D,r_2]}$. Let m_1 resp. m_2 be the instance messages generated by p_1 resp. p_2 in round $r_1 - 2D$ resp. round $r_2 - 2D$:

(*Case 1a*) $r_1 \leq r_2 - D$: Let m_ℓ be the leader message generated by p_1 in round r_1. By Lemma 6.5.4, we know that p_2 must have received some message $m_1' \geq m_\ell$ by round r_2. By assumption, p_2 could not have passed Line 15, therefore, $m_2 > m_1$. Since p_1 remains inside the region until round r_2, however, it follows again by Lemma 6.5.4 that p_1 must have also received some message $m_2' \geq m_2 > m_\ell$ by round r_2, causing p_1 to reset its *leader* variable and thus yielding a contradiction.

(*Case 1b*) $r_1 > r_2 - D$: Since $r_1 - 2D \leq r_2 - 2D$, it follows by Lemma 6.5.3.(b) that nodes p_1 and p_2 are in $\Pi_A^{(r_2-2D,r_2-D]}$. Then, because of Lemma 6.5.4, we know that either p_1 or p_2 passes the if-condition in Line 15 in some round in $(r_2 - 2D, r_2 - D]$. Applying Lemma 6.5.3.(a) provides a contradiction.

(*Case 2*) p_1 used Line 43 and p_2 used Line 20: This implies that p_2 received a leader message from some distinct node v_2. Let r_v be the most recent round before round r_2 in which v_2 considers itself as the leader, i.e., $r_v < r_2$ and $v_2 \in \Pi_E^{r_v}$. Recall the assumption that the first violation of agreement happened in round r_2. It follows that $r_v < r_1$, otherwise this violation would have already taken place in round r_v. We distinguish several cases, depending on when r_v takes place:

(*Case 2a*) $r_1 - 2D > r_v$: Since $r_2 \geq r_1$ we have $r_2 - 2D > r_v$. Considering that node v_2 does not generate any leader messages during (r_v, r_2), it follows by the fact that p_2 received such a message in r_2 that p_2 sets its timer $leaderT$ according to the age field of v_2's leader message m_v. Clearly we have $m_v.age > 2D$, and thus expired($leaderT$) holds in round r_2, causing p_2 to reset its $leader$ variable to $\perp$, a contradiction.

(*Case 2b*) $r_1 - D \geq r_v$: By (Case 2a) we know that $r_v \geq r_1 - 2D$, i.e., $r_v \in [r_1 - 2D, r_1 - D]$. Since v_2 is in $\Pi_E^{r_v}$, applying Lemma 6.5.4 shows that p_1 receives some message m' that is at least as high as v_2's leader message m in some round $r' \in (r_v, r_1]$. By assumption, p_1 does not pass the if-condition in Line 15, therefore p_1's instance message $m_1 > m'$. On the other hand, we know that p_2 does pass Line 15 in round r_2 by receiving m, i.e., $m_1 > m' \geq m$. Since p_1 sends m_1 in round $r_1 - 2D$, Lemma 6.5.4 tells us that p_2 must have received a message that is at least as great as m_1 before round r_2. Therefore, p_2 cannot pass Line 15 in round r_2 by receiving m.

(*Case 2c*) $r_1 - D < r_v$: We know that $r_v < r_1$, which means that $r_v \in (r_1 - D, r_1)$. The proof is similar to (Case 2b). In particular, it follows that the instance message m_1 must be greater than the leader message m of v_2, which in turn means that p_2 cannot have pass Line 15 in round r_2 by receiving m.

(*Case 3*) p_1 used Line 20 and p_2 used Line 43: This implies that p_1 received a leader message m_v from node v_1. Let r_v be the most recent round before r_1 that v_1 considers itself as the leader, i.e., $r_v < r_1$. We further distinguish two cases:

(*Case 3a*) $r_1 \leq r_2 - D$: Note that, by Lemma 6.5.4, p_2 must have received some message $m' \geq m_v$ by round r_2, but since p_2 does not pass the if-condition in Line 15, it must be that $m_2 > m' \geq m_v$. Moreover, p_1 receives m_2 before round r_2, and since p_1 still has $leader_{p_1}^{r_2} = v_1$, p_1 does not pass Line 15 during $(r_1, r_2]$, contradicting the fact that $m_2 > m_v$.

(*Case 3b*) $r_1 > r_2 - D$: First, assume that $r_v \leq r_2 - D$: By Lemma 6.5.4, we know that p_2 receives some message $m' \geq m_v$ by round r_2, and, by Lemma 6.5.3.(a), we have $m_2 > m' > m_v$. Moreover, p_1 receives some message $m'' \geq m_2 > m_v$ by round r_1, thus it cannot be that $leader_{p_1}^{r_2} = v_1$. For the case where $r_v > r_2 - D$, we can use an analogous argument as in (Case 2c).

(*Case 4*) p_1 and p_2 both used Line 20: This implies that there are distinct nodes v_1 and v_2 that generate leader messages in rounds $r_1'(\leq r_1 - 1)$ and $r_2'(\leq r_2 - 1)$, respectively. We assume that $r_1' \leq r_2'$; the case $r_1' > r_2'$ follows analogously. Note that p_1 receives the leader message $\langle L, a_1, \_, v_1 \rangle$ from v_1 in round r_1. Clearly it holds that $r_2' - r_1' < 2D$, otherwise p_1 would have executed `startInstance`() in Line 54 and updated its leader variable during $(r_1, r_2]$, therefore violating the assumption that r_1 was the last update before r_2. Thus we can restrict our treatment to two cases:

(*Case 4a*) $r_1' \in (r_2' - 2D, r_2' - D)$: We know that $leader_{v_2}^{r_1'} \neq v_2$, since we assumed that the *first* violation of agreement happened at $r_2 > r_1'$. Considering that $leader_{v_2}^{r_2'} = v_2$, it follows that the *electT* timer of node v_2 must expire in some round $r_2'' \in (r_1', r_2']$. It follows that the instance message in $toSend_{v_2}$ must be greater than the leader message generated by v_1 in round r_1', otherwise r_2 would have passed Line 15, contradicting Lemma 6.5.3.(a). But this directly contradicts Corollary 6.5.6.

(*Case 4b*) $r_1' \in [r_2' - D, r_2')$: By Lemma 6.5.3.(b), we know that $v_1 \in \Pi_E^{r_1' - 2D}$ and $v_1 \in \Pi_A^{(r_1' - 2D, r_1']}$.

First suppose that both, v_1 and v_2 generate their respective first leader message in round r_1' resp. r_2'. Then, by Lemma 6.5.4 it follows that one of them will receive a message greater than or equal to the instance message of the respective other node, and thus pass Line 15 during $[r_2' - D, r_2')$, a contradiction.

If both, v_1 and v_2 have also generated a leader message in $r_1' - D$ resp. $r_2' - D$, we immediately have a contradiction to the assumption that r_2 is the earliest round when

agreement is violated. Now consider the case where v_2 also generates a leader message in $r_2'-D$, while v_1 still has $leader_{v_1}^r \neq v_1$ for $r \in [r_1', r_1'-2D]$. It follows that v_1 receives the message that was in $toSend_{v_2}^{r_2'-2D}$ before round r_1'. If $toSend_{v_2}^{r_2'-2D}.type = L$, we know by Corollary 6.5.6 that v_1 will pass Line 15, which contradicts Lemma 6.5.3. On the other hand, if $toSend_{v_2}^{r_2'-2D}.type = I$ and v_1 does not pass Line 15, then, by Lemma 6.5.4, v_2 will receive some message equal or greater to v_1's instance message and pass Line 15, a contradiction. □

6.5.6.2 Stability

Lemma 6.5.9. *Suppose that $leader_{p_i}^r = p_i$. Then, at round r, p_i does not run its waitT nor its electT timer, i.e., $\neg\texttt{running}(waitT)_{p_i}^r \wedge \neg\texttt{running}(electT)_{p_i}^r$ hold.*

Proof. Note that the only place where p_i sets its leader variable to its *own* id is in Line 43. First, assume that $\texttt{running}(waitT)_{p_i}^r$ holds. Observe that timer *waitT* is only set in Line 10 upon p_i entering the region in some round $r_1 < r$. Moreover, p_i does not start any other timer until *waitT* has expired. Thus p_i does not pass Line 41 during $[r_1, r]$, a contradiction.

Now, suppose that $\neg\texttt{running}(electT)_{p_i}^r$ holds. In that case p_i must have executed $\texttt{startTimer}(electT, 2D)$ in Line 62, which means that p_i has reset its leader variable in Line 58. Again, p_i does not pass Line 41 before round r, which provides the contradiction. □

The next lemma shows that if a node claims to be the leader, it continues to do so until it crashes or leaves the region.

Lemma 6.5.10. *Suppose that $leader_{p_i}^{r_1} = p_i$ and $p_1 \in \Pi_A^{(r_1,r_2]}$. Then it holds for all $r \in (r_1, r_2]$ that $leader_{p_i}^r = p_i$.*

Proof. Assume for the sake of a contradiction that there is a round $r \in (r_1, r_2]$ such that $leader^r_{p_i} \neq p_i$; choose r such that it is the earliest round after r_1 where this happens.

First, suppose that p_i updated its leader variable in Line 58. Lemma 6.5.9 tells us that p_i does not pass the if-checks in Lines 29 or 41, and since $leader^{r-1}_{p_i} = i$, node p_i does not execute Line 54 in round r; therefore p_i does not call `startInstance`() in round r.

Observe that p_i generates a leader message at some round r' where $r - r' < D$, since r is the earliest round after r_1 where p_i is not a leader. Then, Corollary 6.5.6 and (*Agreement*) provides a contradiction

Now, suppose that p_i executed Line 20 in r by receiving a leader message m_ℓ that was generated by itself before entering the region for the last time before r. Considering that p_i contains a newer leader message in $toSend^r_{p_i}$, this yields a contradiction to the priority relation on leader messages, since $m_\ell < toSend^r_{p_i}$ and thus p_i does not pass the if-condition in Line 15. Thus, if p_i executed Line 20, it must be that p_i received a leader message $m_k = \langle L, \_, \_, p_k \rangle$ in round r that was generated by some distinct node p_k in round r_k where

$$m_k.attNum > attemptNum^r_{p_i}. \tag{6.3}$$

By the agreement property (Lemma 6.5.8), we know that $r_k < r_1$ because p_k only generates leader messages when $leader^{r_k}_{p_k} = p_k$. Together with the fact that some node broadcasts m_k in all rounds in $[r_k, r)$, the D-connectedness assumption yields upper and lower bounds on r_k, i.e., $r_k \in (r_1 - D, r_1)$. Note that Lemma 6.5.3.(b) tells us that $p_i \in \Pi_A^{(r_1-2D, r_1]}$. Again, applying the agreement property shows that $leader^{r_k}_{p_i} \neq p_i$, thus the $electT$ of p_i must have expired during $(r_k, r_1]$, for $leader^{r_1}_{p_i} = p_i$

to hold. If p_k has previously generated a leader message in $r_k - D$, or if $r_1 - r_k \geq D$, a leader message of p_k will be received by p_i before r_1, contradicting (6.3). Thus we can assume that $r_1 - r_k < D$. If p_k has $leader_{p_k}^{r_k-1} \neq k$, then the instance message m'_k that is in $toSend_{p_k}^{r_k-1}$ is broadcast during $[r_k - 2D, r_k)$ by p_k and therefore also received by p_i. Since

$$m'_k.attNum = m_k.attNum - 1$$

and p_i increases its variable $attemptNum$ by exactly one before generating the leader message in r_1, it follows that p_i must have passed Line 15, again a contradiction. □

Lemma 6.5.11 (Stability)**.** *The algorithm in Figures 6.1 and 6.2 satisfies property* (*Stability*) *with bound* $B_S = 2D - 1$.

Proof. For the sake of a contradiction, suppose that for rounds r_1 and r_2, where $r_1 < r_2$, it holds that

$$leader_{p_i}^{r_1} = p_j \wedge leader_{p_i}^{r_2} \neq p_j,$$

and assume that $p_i \in \Pi_A^{[r_1,r_2]}$ and $p_j \in \Pi_A^{[r_1-B_S,r_2]}$.

Let $r \in (r_1, r_2]$ be the earliest round where p_i sets $leader_{p_i}^{r}$ such that $leader_{p_i}^{r} \neq p_j$. If $j = i$, Lemma 6.5.10 tells us that p_i must have left the region during $(r_1, r]$; we can therefore restrict ourselves to the case where $j \neq i$. Note that it is sufficient to consider the case where p_i updates its leader variable in either Lines 20, 23 or Line 58, since for setting the leader via Line 43, p_i needs to execute Line 58 first.

Consider the case where p_i used Line 58. Let r_j be the latest round before r_1 where p_j still claims to be the leader. We can assume that either p_i did not receive a leader message from p_j that was generated during $(r - 2D, r]$ or, p_i did receive such a leader message from p_j but subsequently also received a message m_k from some node $p_k \neq p_j$ that enabled it to pass Line 19. In the former case, it follows by

Lemma 6.5.10 and the fact that a leader generates new leader messages in intervals of D rounds that there must be a round after r_j where p_j left the region, yielding a contradiction. In the case where p_i receives message m_k, it follows by the code of the algorithm that r is not the earliest round after r_1 where p_i updates its leader variable, a contradiction.

Now, consider the case where p_i used Lines 20 or 23, i.e., p_i received a message m_k in round r generated by some node p_k in round r_k, where $p_k \neq p_j$ and such that

$$m_k.attNum > toSend^r_{p_i}.attNum. \tag{6.4}$$

If m_k was generated before or at round $r - 3D$, it follows by Lemma 6.5.4 that p_j will pass Line 15 by receiving some message $\geq m_k$ before generating the last leader message and therefore still being inside the region; a contradiction to Lemma 6.5.10. Thus, m_k was generated in round $r_k \in (r - 3D, r)$. Corollary 6.5.6 tells us that $m_k.type = L$, which by Lemma 6.5.8 implies that $r_k \in (r - 3D, r_1)$. If $r_k \geq r_j$, Lemma 6.5.8 implies that p_j must have left the region and we are done. Therefore, consider the case where $r_k < r_j$. If $r_j - r_k > D$, we again have a contradiction to Lemma 6.5.10 due to (6.4); therefore $r_j - r_k < D$. For the case where p_k already generated a leader message in $r_k - D$, the contradiction again follows by (6.4), hence, we can assume that $\texttt{running}(electT)_{p_k}$ holds during $[r_k - 2D, r_k)$ and $toSend_{p_k}.type = I$. If

$$toSend.attNum^{r_k-1}_{p_k} > toSend.attNum^{r_k-1}_{p_j},$$

we have a contradiction since p_j will pass Line 15 before r_j; otherwise, it must be that

$$toSend.attNum^{r_k-1}_{p_k} = toSend.attNum^{r_k-1}_{p_j}.$$

This, however, implies that $toSend.type_{p_j}^{r_k-1} = L$, and, since p_k will receive this leader message before r_j, it will not generate a leader message itself, a contradiction. □

6.5.6.3 Termination

We say that node p *expired* message m at round r, if p expired its $leaderT$ timer at r and called `startInstance`() while having m in its $toSend$ variable just before the generation of a new message. In addition, we say that a message m *caused* the generation of instance message m' at round r, if there exists a node p such that it expired message m at round r and as a consequence p generated m' in round r.

The next lemma is immediate from the code.

Lemma 6.5.12. *A node stays live at least* $5D$ *rounds in* R *before starting/expiring any timer other than the* $waitT$ *timer. In other words, it requires at least* $5D$ *rounds to expire the* $waitT$ *timer.*

Lemma 6.5.13. *Suppose some node* p *expired message* m *at round* r*. Then the following holds:*

(a) If $m.type = L$ *(i.e.* m *is a leader message),* m *must have been generated before or at* $r - 2D$*.*

(b) If $m.type = I$ *(i.e.* m *is an instance message),* m *must have been generated before or at* $r - 3D$*.*

Proof. (a) Suppose, in contradiction, that leader message m was generated during $(r - 2D, r]$. If m was generated by p, then Line 54 will simply not be executed since Line 50 will become true. If m was not generated by p, then, by Line 21, p must not have expired its $leaderT$ timer by r since $leaderT$ is set to expire exactly $2D$ rounds after the generation of the leader message.

(b) Suppose, in contradiction, that instance message m was generated during $(r -$

$3D, r]$. Let $r' \in (r - 3D, r]$ be the earliest round where $toSend_p^{[r',r)} = m$. We divide into two cases:

(*Case 1*) Instance message m was generated by p: In this case, $p \in \Pi_A^{[r',r]}$ and p must not be running its $waitT$ timer during $[r', r]$ because otherwise p must have reentered region R and Lemma 6.5.12 prevents p from expiring m at r. If p generated m at $r'' \in (r - 3D, r')$, then Lemma 6.5.2 implies that $toSend_p^{[r'',r)} = m$ which contradicts the fact that r' is the earliest round where $toSend_p^{[r',r)} = m$. So, r' is the round when p generated m. We further divide into two cases:

(*Case 1a*) $r' \leq r - 2D$: By the code, Line 62 is executed when m is generated. Also, Lemma 6.5.2 tells us that Line 15 does not become true until round r. So, by the code, $electT$ timer will expire with Line 42 being true causing the generation of a leader message m' by round r where $m' > m$. Hence, $toSend_p^r \neq m$.

(*Case 1b*) $r' > r - 2D$: Same as in (*Case 1a*), Line 62 is executed when m is generated and Line 15 will not become true until round r. Hence, by the code, $electT$ timer will still be running at round r at p since $electT$ timer is set to expire $2D$ rounds after the generation of message m.

(*Case 2*) Instance message m was not generated by p: We further divide into two cases:

(*Case 2a*) Node p is running its $waitT$ timer at round r': Since p expires m at round r, there exists a round $r'' \in (r', r]$ where p stops running its $waitT$ timer. By the code, p executes Line 40 at r''. Let $r_1 \in (r - 3D, r]$ be the round when m was generated. The execution of Line 40 sets p's $electT$ timer to expire at round $r'' + 2D$ and after the $electT$ timer expires, by Line 48, an additional D rounds is needed to expire the $leaderT$ timer. Hence, p does not expire m at round r.

(*Case 2b*) Node p is not running its $waitT$ timer at round r': The proof is similar to (*Case 2a*) except that Line 28 is executed instead of Line 40. □

Lemma 6.5.14. *Suppose instance message m was generated by p at round r. Let m' be the message that caused the generation of m at p at round r ($m' < m$). Then, for all $r_q \in [r-D, r+D]$, any node $q \in \Pi_A^{[r_q-D,r_q]}$ has in $toSend_q^{r_q}$ either m' or an instance message generated by expiring m'.*

Proof. Suppose not. We divide into two cases:
(*Case 1*) For some $r_q \in [r-D, r]$, some node $q \in \Pi_A^{[r_q-D,r_q]}$ has a message m_1 in $toSend_q^{r_q}$ where $m_1 \neq m'$ and m_1 was not generated by expiring m': By Lemma 6.5.13, m' must have been generated before or at $r-2D$ and by Assumption 6.3.1 and Lemma 6.5.4, m' is the highest priority instance message in R until round $r-D$. Message m_1 must have been generated during $(r-D, r]$ because otherwise, by Assumption 6.3.1, Lemma 6.5.12, and Lemma 6.5.4, p receives m_1 or some message that has higher priority than m_1 by round r if $m_1 > m'$ and q receives m' or some message that has higher priority than m' by round $r-D$ otherwise. A contradiction.

Now, suppose m_1 is a leader message. Let p_1 be the node that generated m_1. Then, p_1 must have generated an instance message m'' at round $r_1 \in (r-3D, r-2D]$ since it requires exactly $2D$ rounds for leader message m_1 to be generated after the generation of m''. By Assumption 6.3.1, Lemma 6.5.12, and Lemma 6.5.4, p receives m'' or some message $> m''$ by round r if $m'' > m$, and p_1 receives m' or some message that has higher priority than m' by $r-D$ otherwise. A contradiction.

From the above, we can restrict m_1 to be an instance message generated during $(r-D, r]$. Then, there exists a message m_2 that caused the generation of m_1 at some node q' at round $r' \in (r-D, r]$. Message m_2 must be $m_2 < m'$ because otherwise, by Assumption 6.3.1, Lemma 6.5.12, and Lemma 6.5.4, q receives m_2 or some message that has higher priority than m_2 by round r which yields a contradiction.

Now, by Lemma 6.5.13, m_2 must be generated before or at round $r-2D$. But,

by round r', node q' must receive m' or some message that has higher priority than m' (again by Assumption 6.3.1, Lemma 6.5.12, and Lemma 6.5.4), Hence, m_1 does not exist, a contradiction.

(*Case 2*) For some $r_q \in (r, r+D]$, some node $q \in \Pi_A^{[r_q - D, r_q]}$ has a message m_1 in $toSend_q^{r_q}$ where $m_1 \neq m'$ and m_1 was not generated by expiring m': By (*Case 1*), m_1 must have been generated during $(r, r+D]$. The rest of the proof is similar to (*Case 1*). □

Lemma 6.5.15. *Suppose instance message m was generated by p at round r. Let m' be the message that caused the generation of m at p at round r ($m' < m$). Then, for all $r_q \in [r-D, r+D]$, any node q that is live, in R, and not running its waitT timer at r_q has in $toSend_q^{r_q}$ either m' or an instance message generated by expiring m'.*

Proof. The proof is direct from Lemma 6.5.12 and Lemma 6.5.14. □

Lemma 6.5.16. *For any two instance messages m_1 and m_2 that were generated by p_1 and p_2 by expiring the same message m at rounds r_1 and r_2, respectively, $|r_1 - r_2| < D$ holds.*

Proof. First note that both m_1 and m_2 have higher priority than m, and $m_1.attNum = m_2.attNum$. Without loss of generality, suppose m_1 has higher priority than m_2 (p_1 entered R ealier than p_2 or p_1 and p_2 entered R at the same round) and $(r_2 - r_1) \geq D$. By Assumption 6.3.1, Lemma 6.5.12, and Lemma 6.5.4, p_2 receives m_1 or some message that has higher priority than m_1 by round r_2. Hence, by Lemma 6.5.2, p_2 will not expire m, a contradiction. □

Lemma 6.5.17. *Suppose message m' caused the generation of instance message m ($m > m'$) at node p at round r. Let m_1 be the highest priority instance message that*

was generated by any node by expiring m' and let p_1 be the node that generated m_1. Also, let r_1 be the round when m_1 was generated. Then, the following holds:

(a) If $p_1 \in \Pi_E^{r_1+2D}$ and p_1 is not running its waitT timer at r_1+2D, then p_1 elects itself as the leader at round r_1+2D.

(b) For each node $q(\neq p_1)$ that is not running its waitT timer at r_1+3D and $q \in \Pi_E^{r_1+3D}$, it holds for all $r' \in [r_1+2D, r_1+3D]$ that $toSend_q^{r'} = m_1$, if p_1 did not elect itself as the leader.

(c) For any node $q(\neq p_1)$ that is not running its waitT timer at r_1+2D and $q \in \Pi_E^{r_1+2D}$, it holds that q expires its electT timer at round r_1+2D and sets its leaderT timer to expire at r_1+3D.

Proof. (a) Suppose, in contradiction, that p_1 did not elect itself as the leader at round r_1+2D. This means that p_1 received a message m_2 during $[r_1, r_1+2D]$ where $m_2 > m_1$. By Lemma 6.5.15 and the fact that m_1 was the highest priority instance message that was ever generated by expiring m', m_2 must have been generated during (r_1+D, r_1+2D) by expiring a message m'' that was in turn generated by expiring m'. By Lemma 6.5.16, the earliest round that m'' can be generated is r_1-D. Hence, Lemma 6.5.13 implies that m'' cannot become $\geq 3D$ rounds old (i.e., cannot expire) until round r_1+2D. A contradiction.

(b) Suppose, in contradiction that for some $r_2 \in [r_1+2D, r_1+3D]$, $toSend_q^{r_2} = m_2 \neq m_1$. Let r_2 be the round when m_2 was generated by some node p_2. By Lemma 6.5.15 and the fact that m_1 was the highest priority instance message that was ever generated by expiring m', m_2 must have been generated at $r'' \in (r_1+D, r_1+3D]$. We divide into two cases:

(*Case 1*) m_2 is a leader message: Then, p_2 must have generated an instance message m_2' at round $r''-2D \in (r_1-D, r_1+D]$ and $toSend_{p_2}^{r''} = m_2'$ holds before p_2

called `startInstance`(). By Lemma 6.5.15, m'_2 must be the same as either m' or some message generated by expiring m'. By Assumption 6.3.1, Lemma 6.5.12, and Lemma 6.5.4, p_2 receives m_1 or some message that has higher priority than m_1 by round $r'' - D (< r_1 + 2D)$, since m_1 is the highest priority instance message generated by a node by expiring m'. Hence, by Lemma 6.5.2, $toSend_{p_2}^{r''} > m'_2$, a contradiction. (*Case 2*) m_2 is an instance message: Then, there exists a message m'_2 that expired at p_2 at r''. By Lemma 6.5.13, m'_2 must have been generated before or at $r'' - 2D (< r_1 + D)$. By Lemma 6.5.15 and the fact that m_1 was the highest priority instance message that was ever generated by expiring m', we have $m'_2 < m_1$. Hence, by Assumption 6.3.1, Lemma 6.5.12, and Lemma 6.5.4, p_2 receives m_1 or some message that has higher priority than m_1 by round r'' preventing m'_2 to expire at p_2 at round r'' (by Lemma 6.5.2). A contradiction.

(c) Suppose q does not expire its *electT* timer at $r_1 + 2D$. This means that $toSend_q^{r_1+2D}$ does not contain m_1 because otherwise Line 28 enforces q to set its *electT* timer to expire at round $r_1 + 2D$. Assumption 6.3.1 and Lemma 6.5.15 implies that, by round $r_1 + D$, q receives m_1 and $toSend_q^{r_1+D} = m_1$. So, in order for q to not expire its *electT* timer at $r_1 + 2D$, q must receive some message $m_2 > m_1$ during $(r_1 + D, r_1 + 2D]$. The rest of the proof is similar to (*Case 1*) and (*Case 2*) of *(b)*. □

Lemma 6.5.18. *Suppose node q entered R at round r. Consider the three rounds $r+5iD$, $r+5(i+1)D$, and $r+5(i+2)D$, for $i \geq 2$, and assume that waitT is running for all $r' \in [r + 5iD, r + 5(i + 2)D)$. If q decides not to stop running waitT at round $r + 5(i + 2)D$, then at least one node that was live, inside R, and not running its waitT timer crashed or left region R during $[r + 5(i - 1)D - D, r + 5(i + 2)D]$.*

Proof. For some $k(\geq 2)$, let m_1, m_2, and m_3 be the message contained in $toSend_q^{r+5kD}$, $toSend_q^{r+5(k+1)D}$, and variable $toSend_q^{r+5(k+2)D}$, respectively. Suppose that all nodes

that are still live, in R, and not running their $waitT$ timer do not crash or leave region R during $[r + 5(k-1)D - D, r + 5(k+2)D]$. We divide into five cases:

(*Case 1*) m_1 is a leader message and m_2 is an instance message: In this case, m_1 must have been generated during $(r + 5(k-1)D - D, r + 5kD]$ because otherwise, by Assumption 6.3.1, q receives m_1 by round $r + 5(k-1)D$ and as a consequence `differ`$(prevToSend, toSend)$ will return false causing q to stop running its $waitT$ timer at round $r + 5kD$. This means that there exists a node q' that elected itself as the leader during $(r+5(k-1)D-D, r+5kD]$. Applying (stability) and Corollary 6.5.6 on q tells us that q crashed or left the region during $(r+5(k-1)D-D, r+5(k+1)D]$. A contradiction.

(*Case 2*) m_1 is a leader message and m_2 is a leader message ($m_1 \neq m_2$): The proof of this case is similar to (*Case 1*).

(*Case 3*) m_1 is an instance message and m_2 is an instance message: Similar to (*Case 1*), m_1 must have been generated during $(r + 5(k-1)D - D, r + 5kD]$. This means that there exists a message m' that caused the generation of m_1 at $r' \in (r + 5(k-1)D - D, r + 5kD]$. By Lemma 6.5.17.(a), the node, say q'', that generated the highest priority instance message m'' by expiring m' elects itself as the leader and by Lemma 6.5.16, the round that q'' elects itself as the leader cannot be later than $r + 5kD + 3D$ (D rounds for the difference between the generation round of m_1 and m'', and $2D$ rounds for m'' electing itself as the leader). Again, applying (stability) and Corollary 6.5.6 tells us that q must have crashed or left the region during $(r + 5(k-1)D - D, r + 5(k+1)D]$, A contradiction.

(*Case 4*) m_1 is an instance message, m_2 is a leader message, and m_3 is a leader message: Simply applying (*Case 2*) on m_2 and m_3 gives us a contradiction.

(*Case 5*) m_1 is an instance message, m_2 is a leader message, and m_3 is an instance message: Applying (*Case 1*) on m_2 and m_3 provides us with a contradiction. □

Lemma 6.5.19. *Suppose that node p stops running its waitT timer at round r. If $p \in \Pi_A^{[r,r+4D]}$, p is not running its waitT timer during $[r, r+4D]$, and p does not elect a leader other than itself during $[r, r+4D]$, then there exists a round $r_p \in [r, r+4D]$ such that p generates an instance message.*

Proof. We distinguish the following four cases:

(*Case 1*) An expired ($2D$ rounds old or older) leader message was in $toSend_p^r$: In this case, Line 36 to Line 38 will be executed and in turn Line 49 will evaluate to true at round r. Since Line 36 prevents p from electing itself as the leader, `startInstance`() will be called at Line 54. Hence, p generates an instance message by executing Line 61 at round r.

(*Case 2*) An expired ($3D$ rounds old or older) instance message was in $toSend_p^r$: In this case, Line 40 will be executed and in turn Line 41 evaluates to true. Also, Line 42 evaluates to false since initially $candidate = \perp$. So, Line 48 will be executed and in turn Line 49 will evaluate to true at round r. Now, since initially $leader = \perp$, Line 50 will evaluate to false causing Line 54 to be executed. Hence, p generates an instance message by executing Line 61 at round r.

(*Case 3*) A non-expired (less than $2D$ rounds old) leader message was in $toSend_p^r$: Let m be the leader message in $toSend_p^r$. If m was not generated by p, then Line 37 is executed at round r which contradicts the assumption that p does not elect a leader other than itself during $[r, r+4D]$. If m was generated by p, then Lemma 6.5.12 implies that m was generated before round $r-5D$ which contradicts the assumption that m is less than $2D$ rounds old.

(*Case 4*) A non-expired (less than $3D$ rounds old) instance message was in $toSend_p^r$: Let m be the instance message in $toSend_p^r$ and let m' be the message that caused the generation m. Also, let $r' \in (r-3D, r]$ be the round when m was generated

(since m did not expire at round r, m must have been generated during $(r-3D, r]$). By Lemma 6.5.16, the highest priority instance message, say m_1, that was ever generated by expiring m' is generated at $r_1 \in (r'-D, r'+D)$. Also, by Lemma 6.5.17.(a) and Lemma 6.5.17.(b), for each node $q \in \Pi_E^{r_1+3D}$, $toSend_q^{r_1+3D} = m_1$ and the $leaderT$ timer expires at r_1+3D. Hence, `startInstance`() is executed (Line 54) at $r_1+3D \in (r-D, r+4D)$. If $r_1+3D \in (r, r+4D)$, then we are done. If $r_1+3D \in (r-D, r]$, then, by Assumption 6.3.1 and Lemma 6.5.4, p also must have received m_1 or some message that has higher priority than m_1 by r_1+3D. Variable $toSend_p^r$ cannot have m in it because it will contradict Lemma 6.5.2 since $m < m_1$. □

Lemma 6.5.20. *Suppose that node p generates an instance message m at round r. If $p \in \Pi_A^{[r,r+4D]}$, p is not running its $waitT$ timer during $[r, r+4D]$, and p does not elect a leader other than itself during $[r, r+4D]$, then there exists a round $r_p \in (r, r+4D]$ such that p generates an instance message.*

Proof. The proof is similar to (*Case 4*) of Lemma 6.5.19. □

Lemma 6.5.21. *Suppose node q enters R at round r and there exists a node p that is live, in R, and not running its $waitT$ timer at round r. Then, q does not stop running its $waitT$ timer unless p elects a leader, crashes, or leaves region R.*

Proof. By Lemma 6.5.19 and Lemma 6.5.20, we know that p generates an instance message at least every $4D$ rounds as long as p does not crash, leave the region, or elect a leader. So, by Assumption 6.3.1 and Lemma 6.5.4, q receives the instance message generated by p or some message that has higher priority than it at least every $5D$ rounds. Along with Lemma 6.5.17, the above implies that q receives a new message every $5D$ rounds from a node that entered earlier than itself. Hence, Line 30 will evaluate to true which causes q to start its $waitT$ timer again. □

Lemma 6.5.22 (Termination)**.** *The algorithm in Figures 6.1 and 6.2 satisfies (termination) with bound $B_T = 22D + 14(n-1)D$ where n is the total number of nodes in the system.*

Proof. Suppose that node p entered R at round r_e. For the sake of a contradiction, we assume that $p \in \Pi_E^{r_e}$, $p \in \Pi_A^{(r_e, r_e+22D+14(n-1)D]}$, and for all $r \in [r_e, r_e+22D+14(n-1)D]$, $leader_p^r = \perp$. The worst case bound on B_T can be obtained if the adversary causes p to run its $waitT$ timer as long as possible and, after p stops running its $waitT$ timer, the adversary forces p to delay its decision as long as possible. By Lemma 6.5.18, we know that, after round r_e+10D, at least one node that was not running its $waitT$ timer crashes or leaves region R in every $10D$ rounds if p manages to run its $waitT$ timer continuously during those $10D$ rounds. However, this can happen only at most $n-1$ times because there can be at most $n-1$ nodes not running its $waitT$ timer. Also, Lemma 6.5.18 implies that even in the case where a node q that has left the region (resp. crashed) reenters (resp. recovers from its crash inside R) it will always hold that $enteredSince_q < enteredSince_p$ upon reentering (resp. recovering). Therefore, all newly generated messages m will have $m.entSince < enteredSince_p$ from round $r_e + 10D + 10(n-1)D$ on. Hence, Line 30 evaluates to false at round $r_e + 10D + 10(n-1)D + 5D$ causing timer $waitT$ to stop running.

Now, we consider the worst case for delaying p's decision after p stops running its $waitT$ timer; let r_w be the respective earliest round when $\neg\texttt{running}(waitT)_p^{r_w}$ holds. Note that r_w can be at most $r_e + 15D + 10(n-1)D$. Let r_f be the first round after $r_w + D$ when p generates an instance message. By assumption, we know that p did not elect a leader during $[r_w, r_f + 4D]$. Therefore, Lemma 6.5.19 and Lemma 6.5.20 imply that $r_f \in (r_w + D, r_w + 5D]$, since p generates an instance message every $4D$ rounds if it did not elect a leader. Let m_f be such an instance message

generated by p at round r_f. We know that there exists a message m' that caused the generation of m_f. Let m'_f be the highest priority instance message that was ever generated by expiring m' and let p'_f be the node that generated m'_f. Lemma 6.5.17.(a) tells us that if p'_f does not crash or leaves region R, then p'_f elects itself as the leader by round $r_f + 3D$ (by Lemma 6.5.16, m'_f can be generated as late as $r_f + D$ and it takes $2D$ rounds to elect itself as the leader). To delay p's decision as long as possible, we consider the case of node p'_f crashing or leaving region R after generating m'_f but before electing itself as the leader. Note that even if p'_f reenters R (resp. recovers in R) after leaving R (resp. crashing), the round that p'_f reenters R (resp. recovers in R) cannot be less than r_w since, by Lemma 6.5.16, m_f can only be generated as early as $r_f - D > r_w$. So, as long as p is live, in R, and not running its $waitT$ timer, Lemma 6.5.21 prevents p'_f from stopping its $waitT$ timer after its reentry (resp. recovery). Now, since p'_f did not elect itself as the leader, we know by Lemma 6.5.20 that p will generate a new instance message by round $r_f + 4D$. The above can happen at most $n - 1$ times since there can be at most $n - 1$ nodes that are not running their $waitT$ timer along with p. So, after round $r_f + 4(n-1)D$, p will be the only node that is live, in R, and not running its $waitT$ timer and p will elect itself as the leader by round $r_f + 4(n-1)D + 2D$ (since p is the only node left, it will be the only node generating an instance message). Hence, in the worst case, p elects a leader by round $r_w + 7D + 4(n-1)D$ since r_f is at most $r_w + 5D$.

For node p, the overall worst case would be taking $15D + 10(n-1)D$ rounds for stopping its $waitT$ timer and taking $7D + 4(n-1)D$ for electing a leader after stopping its $waitT$ timer. Summing up shows that in the worst case, by round $r_e + 15D + 10(n-1)D + 7D + 4(n-1)D = r_e + 22D + 14(n-1)D$, node p elects a leader, a contradiction. □

We obtain the following theorem directly from Lemmas 6.5.7, 6.5.8, 6.5.11, and 6.5.22:

Theorem 6.5.23. *The algorithm in Figures 6.1 and 6.2 solves the RCLE problem with bounds* $B_V = B_S = 2D - 1$ *and* $B_T = 22D + 14D(n-1) = \mathcal{O}(nD)$.

6.6 A Condition on Mobility

In this section, we provide a condition on mobility that implies Assumption 6.3.1 with a bounded communication diameter D. Our mobility condition generalizes the mobility condition of [16] in a way such that it no longer depends on a specific coordinate system. We assume region R to be a 2-dimensional convex polygon. In addition, we assume that nodes move at a speed bounded by some constant σ.

We first define some basic properties for our mobility condition. Let δ be a fixed constant such that $0 < \delta \leq C - \Delta\sigma$.[3] Given two positions ϕ_1 and ϕ_2, let $dist(\phi_1, \phi_2)$ be the Euclidean distance between ϕ_1 and ϕ_2.

Definition 6.6.1. *Let* $p_0, p_1, \ldots, p_{n-1}$ *be a sequence of nodes such that* p_i *is at position* ϕ_i *at the beginning of round* $r + i$, $0 \leq i \leq n-1$, *and let* ϕ_n *be any position in* R. *The sequence of positions* $\phi_0, \phi_1, \ldots, \phi_n$ *is called a* proper propagation sequence from p_0 to ϕ_n starting at round r, *denoted as* $P^r_{\phi_0 \to \phi_n}$, *if it holds that*

(a) ϕ_0 *and* ϕ_n *are both in* R,

(b) p_i *broadcasts in round* $r + i$,

(c) p_i *and* p_{i+1} *are live and connected in round* $r+i$, $0 \leq i \leq n-2$, *and throughout round* $r + n - 1$, p_{n-1} *is live and within distance* C *of* ϕ_n,

(d) $\delta \leq dist(\phi_i, \phi_{i+1})$, *and*

[3] Recall from Section 6.3 that C refers to the communication radius and Δ denotes the round duration. We allow a range of values for δ to accommodate a range of mobility patterns.

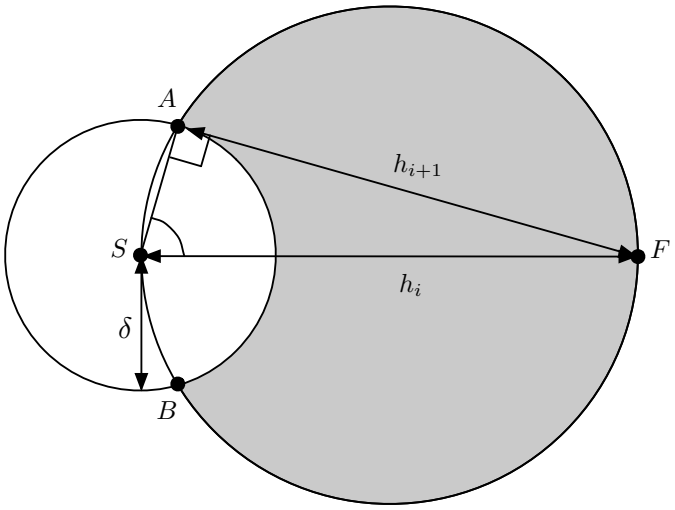


Figure 6.3: Worst case information propagation. Points S and F corresponds to ϕ_i and ϕ_n, respectively. The shaded region of the circle that is uniquely defined by S and F, corresponds to the geographic region in which ϕ_{i+1} must lie. Points A and B represent the worst case for information propagation.

(e) for $0 \leq i \leq n-2$, at the beginning of round $r+i+1$, ϕ_{i+1} lies within R and within a circle that is uniquely defined by positions ϕ_i and ϕ_n, which serve as the two endpoints of the axis of the circle.

Similar to the properties of the well-directed propagation sequence defined in [16], property (d) of Definition 6.6.1 says that information gets closer to its destination in each round and property (e) of Definition 6.6.1 says that as information propagates to its destination, it should reside within a certain geographic sector (see Figure 6.3). Also, note that Definition 6.6.1 corresponds to a JIT path if a real node resides at position ϕ_n. However, notice that Definition 6.6.1 is independent of any specific coordinate system.

The following assumption states our mobility condition:

Assumption 6.6.2 (Mobility Condition)**.** *For every node p_i that is live throughout*

round r and broadcasts a message at the beginning of round r at position ϕ_i in R, and for all positions ϕ_j in R, there exists a proper propagation sequence $P^r_{\phi_i \to \phi_j}$ starting at round r.

Given the above mobility condition, it is possible to obtain the bounded communication diameter D of region R:

Theorem 6.6.3. *Suppose that nodes follow the mobility condition given by Assumption 6.6.2. Then, property D-connectedness is ensured with $D = \lceil \frac{L^2-(C-\Delta\sigma)^2}{\delta^2} \rceil$ where L is the maximum Euclidean distance between any two points in R.*

Proof. The proof is given with the assistance of Figure 6.3. Consider some node p_i that participates in $P^r_{\phi_i \to \phi_n}$, where $0 \leq i \leq n-1$, and suppose that at the beginning of round $r+i$, node p_i broadcasts information I at position S $(=\phi_i)$ which needs to be propagated to position F $(=\phi_n)$ as depicted in Figure 6.3. By properties (d) and (e) of Definition 6.6.1, p_{i+1}, which obtained information I from p_i, has to be in the shaded region of Figure 6.3 at the beginning of round $r+i+1$. Within the shaded region of Figure 6.3, the position farthest away from F is either A or B. This means that, in worst case, ϕ_{i+1} (the position of p_{i+1} at the beginning of round $r+i+1$) can be either A or B. Without loss of generality, suppose $\phi_{i+1} = A$. Let h_i be the Euclidean distance between ϕ_i and ϕ_n. Since angle $\angle SAF$ is a right angle by construction, we obtain the recursive formula

$$h_{i+1}^2 = h_i^2 - \delta^2. \tag{6.5}$$

Since L is the maximum Euclidean distance between any two points in R, we have $h_0 = L$ in the worst case. From this, we can obtain the following closed-form solution to (6.5):

$$h_{i+1} = \sqrt{L^2 - i\delta^2}$$

Now, suppose that $h_{k+1} \leq C - \Delta\sigma$. In this case, p_{k+1} will be within C of ϕ_n throughout round $r + k + 1$ since the maximum distance that a node can move is bounded by $\Delta\sigma$. Applying this to the above closed-form solution yields

$$h_{k+1} = \sqrt{L^2 - k\delta^2} \leq C - \Delta\sigma$$

and therefore

$$\frac{L^2 - (C - \Delta\sigma)^2}{\delta^2} \leq k.$$

Since k is the worst case number of hops for an information I to reach its destination, we can conclude that the bounded communication diameter is $D = \left\lceil \frac{L^2-(C-\Delta\sigma)^2}{\delta^2} \right\rceil$. $\square$

7. CONCLUSION: SUMMARY AND FUTURE WORK

In this section, we summarize our contributions and provide future work.

7.1 Distributed Resource Allocation Algorithms for Static Networks

We have provided two distributed resource allocation algorithms for static networks, specifically, two distributed algorithms that solve the dining philosophers problem with failure locality 1. Since it is impossible to design any failure-locality-1 dining algorithm in asynchronous systems [14], we utilized failure detectors to design our algorithms.

7.1.1 Dining with Bounded Waiting and Failure Locality 1

In Section 2, we provided the formal problem definition of perpetual strong exclusion with bounded waiting and failure locality 1 (BW-$\Box SX$-$FL1$) and presented a message-passing algorithm that solves BW-$\Box SX$-$FL1$ using the local version of the eventually perfect failure detector ($\Diamond P^1$).

Our algorithm carefully combines the doorway technique presented in [68] and the concept of skepticism in [63]. The exclusion property is ensured by a fork/request protocol and the progress property is guaranteed by preventing each node p from entering its doorway if their exists a neighboring node of p that is already in its doorway. The fairness property is ensured by blocking each node p from entering its doorway if any neighbor q of p, while continously trying to access its critical section, gave permission to p to enter its doorway more than a bounded number of times.

In addition to solving BW-$\Box SX$-$FL1$, we provided the first step towards identifying a weakest failure detector for BW-$\Box SX$-$FL1$ by implementing failure detector $\Diamond P^1$ using multiple instances of BW-$\Box SX$-$FL1$.

Our mutual reduction does not necessarily preserve the underlying conflict graph topology; the conflict graph topology considered in one direction of the reduction is not necessarily the same in the other direction. For future work, we would like to identify the weakest failure detector X for solving $BW\text{-}\Box SX\text{-}FL1$ by showing that X and $BW\text{-}\Box SX\text{-}FL1$ are mutually reducible to each other while preserving the underlying conflict graph topology. In addition, we would like to identify weakest failure detectors for variations of the dining philosophers problem that consider failure locality 1.

7.1.2 Stabilizing Dining with Failure Locality 1

In Section 3, we presented a solution for stabilizing (transient fault-tolerant) failure-locality-1 dining considering shared memory systems with regular registers. We utilized the anonymous local eventually perfect failure detector ($?\Diamond P^1$) to obtain our results. Our algorithm borrowed the concept of asynchronous doorways from [13], however, instead of implementing the doorway using a ping/ack protocol (as done in Section 2), we implemented the doorways using stabilizing mutual exclusion subroutines. We also used stabilizing mutual exclusion subroutines to handle fork activities instead of handling it with a fork/request protocol (as done in Section 2). The use of mutual exclusion subroutines makes our algorithm modular.

Notice, in Section 3, we did not specify any $?\Diamond P^1$ implementation. For future work, it will be interesting to seek for an implementation of $?\Diamond P^1$ on partially synchronous shared memory systems using regular registers. In fact, finding an implementation of $\Diamond P^1$ will suffice since $\Diamond P^1$ implies $?\Diamond P^1$. One idea would be to simulate the message passing protocol used for implementing $\Diamond P^1$ in [6, 35] with read/write operations on shared regular registers.

Our algorithm has the following liveness guarantee: each correct hungry process

p that is at least two hops away from any crashed process eventually eats. This property still allows a hungry neighbor of p to overtake p unboundedly many times in accessing the shared resource (the algorithm presented in Section 3 does not have a fairness property as in the algorithm in Section 2). An interesting future work would be to design a stabilizing failure-locality-1 dining algorithm that satisfies bounded waiting: the algorithm should guarantee that is a correct process p only has correct neighbors, then eventually, for any interval in which p is continuously hungry, no neighbor of p eats more than a bounded number of times.

7.2 Distributed Algorithms for Mobile Ad Hoc Networks

We have provided three distributed algorithms for mobile ad hoc networks that can serve as building blocks in designing other useful applications. Specifically, we have presented algorithms for (1) maintaining neighbor knowledge, (2) neighbor detection, and (3) leader election.

7.2.1 Maintaining Neighbor Knowledge in a Road Network

We have presented, in Section 4, a deterministic solution for nodes to maintain neighbor knowledge where nodes communicate via wireless broadcast and are restricted move on a two-dimensional road network. Our solution includes geographical segmentation and constructing a deterministic collision-free broadcast schedule for nodes to exchange neighbor information Considering our broadcast schedule, we have provided a lower bound on the speed of message propagation on a road network.

We relaxed the density requirement and considered dynamic clusters on a road network. We showed that, under a certain condition, neighbor knowledge is maintained when clusters move close to each other. We discussed how we could obtain initial neighbor knowledge using the gossiping algorithm in [30]. We also presented practical values for the parameters that satisfy all constraints in Section 4; this in-

directly shows that the assumptions and constraints in Section 4 do not contradict each other.

Our definition of a road network does not contain any restriction on the intersecting angle between any two lines. This makes our setting applicable to not only urban vehicular networks, where the network topology resembles a perfect grid network, but also suburban vehicular networks.

For future work, it will be interesting to seek for different broadcast schedules that are suitable on two-dimensional space and provide better lower bounds on the speed of message propagation. Another topic for future work would be to provide a definition of two-dimensional cluster (recall that one-dimensional clusters were considered in Section 4) and see if neighbor knowledge can be maintained when two (or more) two-dimensional clusters merge.

7.2.2 Hello-based Neighbor Detection Using the Abstract MAC Layer

In Section 5, we have presented a periodic hello-based neighbor detection algorithm that tolerates bounded clock drift using the abstract MAC layer [45]. The main challenges were to develop a handshaking mechanism between hello message broadcasts and application message broadcasts. Our neighbor detection algorithm generated and broadcast hello messages at the right time by defining pre-hello intervals and blocking application messages to be sent to the abstract MAC layer during each of these intervals. Our neighbor detection algorithm also guaranteed reliable point-to-point communication of application messages by taking into account the following time durations in becoming link up with a nearby node: (1) the time it takes to flush all of the message out of the k-bounded send buffer without any interference, (2) the sum of pre-hello intervals that appear during the course of flushing the k-bounded send buffer, and (3) the bounded message delay for each hello message

that was broadcast during the course of flushing the k-bounded send buffer.

Our algorithm has an advantage of not requiring trajectory information (notice that trajectory functions are used in Section 4). However, there is a trade-off in which the distance between two nodes must be somewhat closer than their communication radius for them to become link up with each other.

For future work, it will be interesting to explore variations of hello message based approaches such as event-based hello (where upper layer messages affect the transmission time of the hello message) and adaptive hello (where hello messages are broadcast depending on the movement speed of a node) [31] instead of periodic hello and compare their performance when used with the abstract MAC layer for neighbor detection.

7.2.3 Regional Consecutive Leader Election

In Section 6, we formally stated the Regional Consecutive Leader Election (RCLE) problem and proved that any algorithm requires $\Omega(Dn)$ synchronous rounds for solving the RCLE problem in the worst case where D is the bounded communication diameter and n is the total number of nodes in the system. Then, we presented and proved correct a fault-tolerant algorithm that solves the RCLE problem. Our algorithm is proven to be asymptotically optimal with respect to the $\Omega(Dn)$ lower bound. In addition, our algorithm solves the RCLE problem with a low message bit complexity per node per round compared to the previous work in [16]. We also introduced a novel condition on the mobility of nodes that ensures appropriate information propagation among the nodes in the region of interest.

By considering a fixed region, leader election is performed among nodes that are relatively close to each other. This is an advantage over other leader election algorithms that consider an arbitrary network diameter. Our system model in Section

6 allows nodes in the region to be temporarily partitioned. This is an advantage over other leader election algorithms that require continuous connectivity.

As mentioned in Section 6, our algorithm is crash fault-tolerant; periodic leader messages enables the detection of a crashed leader. An important open question is how to deal with malicious nodes in our setting; what if a non-leader node generates leader messages to pretend that it is the leader?, what if a non-leader node pretends that it has entered the region the earliest to become the next leader?, etc.

If nodes move in the region in accordance with our condition on mobility, then it is shown that a bounded communication diameter exists. Future work on this topic involves exploring different mobility conditions that are restrictive enough to guarantee a bounded communication diameter while, on the other hand, being flexible enough for real world applications.

REFERENCES

[1] 3rd Generation Partnership Project (3GPP). Radio access network; Multiplexing and multiple access on the radio path (Release 1999). TS 05.02 ver. 8.11.0, 2003.

[2] I. Abraham, D. Dolev, and D. Malkhi. LLS: A locality aware location service for mobile ad hoc networks. In *Proc. of DIALM-POMC Joint Workshop on Foundations of Mobile Computing*, pages 75–84, 2004.

[3] Dana Angluin, James Aspnes, Zo Diamadi, Michael J. Fischer, and Ren Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, 18(4):235–253, 2006.

[4] Gheorghe Antonoiu and Pradip K. Srimani. Mutual exclusion between neighboring nodes in an arbitrary system graph tree that stabilizes using read/write atomicity. In *Proc. of 5th International Euro-Par Conference on Parallel Processing*, volume 1685, pages 823–830, 1999.

[5] Joffroy Beauquier, Ajoy Kumar Datta, Maria Gradinariu, and Frédéric Magniette. Self-stabilizing local mutual exclusion and daemon refinement. *Chicago Journal of Theoretical Computer Science*, 2002(1):1–19, 2002.

[6] Joffroy Beauquier and Synnöve Kekkonen-Moneta. Fault-tolerance and self-stabilization: impossibility results and solutions using self-stabilizing failure detectors. *International Journal of Systems Science*, 28(11):1177–1187, 1997.

[7] Martin Biely, Peter Robinson, and Ulrich Schmid. Weak synchrony models and failure detectors for message passing (k-)set agreement. In *Proc. of 13th In-*

ternational Conference on Principles of Distributed Systems (OPODIS), LNCS 5923, pages 285–299, 2009.

[8] Franois Bonnet and Michel Raynal. Anonymous asynchronous systems: the case of failure detectors. *Distributed Computing*, 26(3):141–158, 2013.

[9] A. Boukerche and K. Abrougui. An efficient leader election protocol for mobile networks. In *Proc. of International Conference on Wireless Communications and Mobile Computing*, pages 1129–1134, 2006.

[10] Sébastien Cantarell, Ajoy Kumar Datta, and Franck Petit. Self-stabilizing atomicity refinement allowing neighborhood concurrency. In *Proc. of 6th International Symposium on Self-Stabilizing Systems (SSS)*, pages 102–112, 2003.

[11] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43:225–267, 1996.

[12] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(4):632–646, 1984.

[13] Manhoi Choy and Ambuj K. Singh. Efficient fault-tolerant algorithms for distributed resource allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(3):535–559, 1995.

[14] Manhoi Choy and Ambuj K. Singh. Localizing failures in distributed synchronization. *IEEE Transactions on Parallel and Distributed Systems*, 7(7):705–716, 1996.

[15] H. C. Chung, P. Robinson, and J. L. Welch. Brief announcement: regional consecutive leader election in mobile ad-hoc networks. In *Proc. of 6th International Workshop on Algorithmic Aspects of Wireless Sensor Networks (Algosensors), LNCS 6451*, pages 89–91, 2010.

[16] H. C. Chung, P. Robinson, and J. L. Welch. Regional consecutive leader election in mobile ad-hoc networks. In *Proc. of DIALM-POMC Joint Workshop on Foundations of Mobile Computing*, pages 81–90, 2010.

[17] H. C. Chung, P. Robinson, and J. L. Welch. Optimal regional consecutive leader election in mobile ad-hoc networks. In *Proc. of 7th ACM SIGACT/SIGMOBILE Workshop on Foundations of Mobile Computing (FOMC)*, pages 52–61, 2011.

[18] H. C. Chung, S. Viqar, and J. L. Welch. Neighbor knowledge of mobile nodes in a road network. In *Proc. of IEEE 32nd International Conference on Distributed Computing Systems (ICDCS)*, pages 486–495, 2012.

[19] Alejandro Cornejo, Saira Viqar, and Jennifer L. Welch. Reliable neighbor discovery for mobile ad hoc networks. *Ad Hoc Networks*, Advance online publication, doi:10.1016/j.adhoc.2012.08.009, 2012.

[20] A. K. Datta, L. L. Larmore, and H. Piniganti. Self-stabilizing leader election in dynamic networks. In *Proc. of 12th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS), LNCS 6366*, pages 35–49, 2010.

[21] Carole Delporte-Gallet, Stephane Devismes, and Hugues Fauconnier. Robust stabilizing leader election. In *Proc. of 9th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS), LNCS 4838*, pages 219–233, 2007.

[22] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Petr Kouznetsov. Mutual exclusion in asynchronous systems with failure detectors. *Journal of Parallel and Distributed Computing*, 65(4):492–505, 2005.

[23] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Andreas Tielmann. The weakest failure detector for message passing set-agreement. In *Proc. of 22nd International Symposium on Distributed Computing (DISC), LNCS 5218*, pages 109–120, 2008.

[24] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, 1971.

[25] Shlomi Dolev and Ted Herman. Dijkstras self-stabilizing algorithm in unsupportive environments. In *Proc. of 5th International Workshop on Self-Stabilizing Systems*, pages 67–81, 2001.

[26] Shlomi Dolev, Ronen I. Kat, and Elad M. Schiller. When consensus meets self-stabilization. *Journal of Computer and System Sciences*, 76(8):884 – 900, 2010.

[27] F. Ellen, S. Subramanian, and J. L. Welch. Maintaining information about nearby processors in a mobile environment. In *Proc. of 8th International Conference on Distributed Computing and Networking (ICDCN)*, pages 193–202, 2006.

[28] Michael Fischer and Hong Jiang. Self-stabilizing leader election in networks of finite-state anonymous agents. In *Proc. of 10th International Conference on the Principles of Distributed Systems*, pages 395–409, 2006.

[29] Z. Fu, P. Zerfos, H. Luo, S. Lu, L. Zhang, and M. Gerla. The impact of multihop wireless channel on TCP throughput and loss. In *Proc. of 22nd Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, volume 3, pages 1744 – 1753, 2003.

[30] Leszek Gasieniec, Aris Pagourtzis, Igor Potapov, and Tomasz Radzik. Deterministic communication in radio networks with large labels. *Algorithmica*, 47(1):97–117, 2007.

[31] V. C. Giruka and M. Singhal. Hello protocols for ad-hoc networks: Overhead and accuracy tradeoffs. In *Proc. 6th IEEE International Symposium on a World of Wireless Mobile and Multimedia Networks (WoWMoM)*, page 354.361, 2005.

[32] Rachid Guerraoui. Non-blocking atomic commit in asynchronous distributed systems with failure detectors. *Distributed Computing*, 15(1):17–25, 2002.

[33] K. Hatzis, G. Pentaris, P. Spirakis, V. Tampakis, and R. Tan. Fundamental control algorithms in mobile networks. In *Proc. of 11th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 251–260, 1999.

[34] Jaap-Henk Hoepman, Marina Papatriantafilou, and Philippas Tsigas. Self-stabilization of wait-free shared memory objects. *Journal of Parallel and Distributed Computing*, 62(5):818–842, 2002.

[35] Martin Hutle and Josef Widder. On the possibility and the impossibility of message-driven self-stabilizing failure detection. In *Proc. of 7th International Symposium on Self-Stabilizing Systems (SSS)*, pages 153–170, 2005.

[36] Rebecca Ingram, Tsvetomira Radeva, Patrick Shields, Saira Viqar, Jennifer E. Walter, and Jennifer L. Welch. A leader election algorithm for dynamic networks with causal clocks. *Distributed Computing*, 26(2):75–97, 2013.

[37] Prasad Jayanti and Sam Toueg. Every problem has a weakest failure detector. In *Proc. of 27th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 75–84, 2008.

[38] Colette Johnen and Lisa Higham. Fault-tolerant implementations of regular registers by safe registers with applications to networks. In *Proc. of 10th International Conference on Distributed Computing and Networking (ICDCN)*, pages 337–348, 2009.

[39] B. Karp and H. T. Kung. GPSR: greedy perimeter stateless routing for wireless networks. In *Proc. of 6th International Conference on Mobile Computing and Networking (MobiCom)*, pages 243–254, 2000.

[40] Y. B. Ko and N. Vaidya. Geocasting in mobile ad-hoc networks: location-based multicast algorithms. In *Proc. of 2nd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA)*, pages 101–110, 1999.

[41] Y.-B. Ko and N. Vaidya. Location-aided routing (LAR) in mobile ad hoc networks. *Wireless Networks*, 6(4):307–321, 2000.

[42] S. Krishnamurthy, R. Chandrasekaran, N. Mittal, and S. Venkatesan. Brief announcement: Synchronous distributed algorithms for node discovery and configuration in multi-channel cognitive radio networks. In *Proc. 20th International Symposium on Distributed Computing (DISC), LNCS 4167*, pages 572–574, 2006.

[43] S. Krishnamurthy, M. R. Thoppian, S. Kuppa, R. Chandrasekaran, N. Mittal, S. Venkatesan, and R. Prakash. Time-efficient distributed layer-2 auto-configuration for cognitive radio networks. *Computer Networks*, 52(4):831–849, 2008.

[44] F. Kuhn and R. Oshman. Dynamic networks: Models and algorithms. *SIGACT News*, 42(1):82–96, 2011.

[45] Fabian Kuhn, Nancy Lynch, and Calvin Newport. The abstract MAC layer. *Distributed Computing*, 24(3-4):187–206, 2011.

[46] Jeffery C. Line and Sukumar Ghosh. A methodology for constructing a stabilizing crash-tolerant application. In *Proc. of 13th Symposium on Reliable Distributed Systems*, pages 12–21, 1994.

[47] Jeffery C. Line and Sukumar Ghosh. Stabilizing algorithms for diagnosing crash failures. In *Proc. of 13th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 376–376, 1994.

[48] D. Liu. Protecting neighbor discovery against node compromises in sensor networks. In *Proc. of 29th International Conference on Distributed Computing Systems (ICDCS)*, pages 579–588, 2009.

[49] Nancy A. Lynch. Fast allocation of nearby resources in a distributed system. In *Proc. of 12th ACM Symposium on Theory of Computing (STOC)*, pages 70–81, 1980.

[50] N. Malpani, Y. Chen, N. Vaidya, and J. L. Welch. Distributed token circulation in mobile ad hoc networks. *IEEE Transactions on Mobile Computing*, 4(2):154–165, 2005.

[51] N. Malpani, J. L. Welch, and N. Vaidya. Leader election algorithms for mobile ad hoc networks. In *Proc. of 4th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (DIALM)*, pages 96–104, 2000.

[52] S. Masum, A. Ali, and M. Bhuiyan. Asynchronous leader election in mobile ad hoc networks. In *Proc. of International Conference on Advanced Information Networking and Applications*, pages 29–34, 2006.

[53] A. Miller. Gossiping in jail. In *Proc. of 5th International Workshop on Algorithmic Aspects of Wireless Sensor Networks (Algosensors), LNCS 5804*, pages 242–251, 2009.

[54] A. Miller. Gossiping in one-dimensional synchronous ad hoc radio networks. *Master's Thesis, University of Toronto*, 2009.

[55] Masaaki Mizuno and Mikhail Nesterenko. A transformation of self-stabilizing serial model programs for asynchronous parallel computing environments. *Information Processing Letters*, 66(6):285–290, 1998.

[56] Kitae Nahm, Ahmed Helmy, and C.-C. Jay Kuo. TCP over multihop 802.11 networks: issues and performance enhancement. In *Proc. 6th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, pages 277–287, 2005.

[57] M. Nesterenko and A. Arora. Tolerance to unbounded byzantine faults. In *Proc. of 21st IEEE Symposium on Reliable Distributed Systems*, pages 22–29, 2002.

[58] Mikhail Nesterenko and Anish Arora. Dining philosophers that tolerate malicious crashes. In *Proc. of 22nd International Conference on Distributed Computing Systems (ICDCS)*, pages 172–179, 2002.

[59] Mikhail Nesterenko and Anish Arora. Stabilization-preserving atomicity refinement. *Journal of Parallel and Distributed Computing*, 62(5):766–791, 2002.

[60] V. Park and M. Corson. A highly adaptive distributed routing algorithm for mobile ad hoc networks. In *Proc. of 16th Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, pages 1405–1413, 1997.

[61] P. Parvathipuram, V. Kumar, and G.-C. Yang. An efficient leader election algorithm for mobile ad hoc networks. In *Proc. of 1st International Conference*

on Distributed Computing and Internet Technology, LNCS 3347, pages 32–41, 2004.

[62] Scott Pike, Yantao Song, and Srikanth Sastry. Wait-free dining under eventual weak exclusion. In *Proc. of 9th International Conference on Distributed Computing and Networking (ICDCN)*, pages 135–146, 2008.

[63] Scott M. Pike and Paolo A.G. Sivilotti. Dining philosophers with crash locality 1. In *Proc. of 24th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 22–29, 2004.

[64] A. Rao, S. Ratnasamy, C. Papadimitriou, S. Shenker, and I. Stoica. Geographic routing without location information. In *Proc. of 9th International Conference on Mobile Computing and Networking (MobiCom)*, pages 96–108, 2003.

[65] Srikanth Sastry, Scott M. Pike, and Jennifer L. Welch. The weakest failure detector for wait-free dining under eventual weak exclusion. In *Proc. of 21st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 111–120, 2009.

[66] Srikanth Sastry, Jennifer L. Welch, and Josef Widder. Wait-free stabilizing dining using regular registers. In *Proc. of 16th International Conference on Principles of Distributed Systems (OPODIS)*, pages 284–299, 2012.

[67] P. A. G. Sivilotti, S. M. Pike, and N. Sridhar. A new distributed resource-allocation algorithm with optimal failure locality. In *Proc. of 12th International Conference on Parallel and Distributed Computing Systems (PDCS)*, pages 524–529, 2000.

[68] Y. Song and S. M. Pike. Eventually k-bounded wait-free distributed daemons. In *Proc. of 37th IEEE/IFIP International Conference on Dependable Systems*

and Networks (DSN), pages 645–655, 2007.

[69] Yantao Song, Scott Pike, and Srikanth Sastry. The weakest failure detector for wait-free, eventually fair mutual exclusion. Technical Report 2007-2-2, Department of Computer Science and Engineering, Texas A&M University, College Station, TX 77843, USA, 2007.

[70] S. Subramanian. Deterministic knowledge about nearby nodes in a mobile one dimensional wireless environment. *Master's Thesis, Texas A&M University*, 2006.

[71] S. Vasudevan, J. Kurose, and D. Towsley. Design and analysis of a leader election algorithm for mobile ad hoc networks. In *Proc. of IEEE International Conference on Network Protocols*, pages 350–360, 2004.

[72] S. Vasudevan, D. Towsley, and D. Goeckel. Neighbor discovery in wireless networks and the coupon collectors problem. In *Proc. of 15th International Conference on Mobile Computing and Networking (MobiCom)*, pages 181–192, 2009.

[73] Saira Viqar and Jennifer L. Welch. Deterministic collision free communication despite continuous motion. *Ad Hoc Networks*, 11(1):508–521, 2013.